PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :
C12Q 1/68, 1/70, C12P 19/34, C07H 21/02, 21/04, C25B 1/00, 7/00, B01D 61/42, 61/44, C25D 13/00, G01N 27/26

A1

(11) International Publication Number: WO 96/35810

(43) Internati nal Publication Date: 14 November 1996 (14.11.96)

(21) International Application Number: PCT/US96/06579

(22) International Filing Date: 9 May 1996 (09.05.96)

(30) Priority Data:
438,231 9 May 1995 (09.05.95) US

(71) Applicant: CURAGEN CORPORATION [US/US]; 322 East Main Street, Branford, CT 06405 (US).

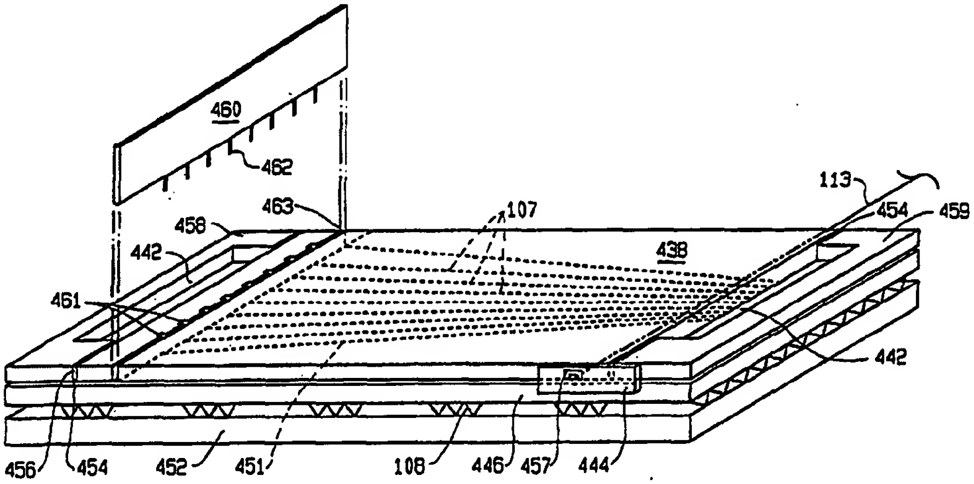
(72) Inventors: SIMPSON, John, W.; 53 Brushy Plain Road #3B, Branford, CT 06405 (US). ROTHBERG, Jonathan, M.; 45 B Cocheco Avenue, Branford, CT 06405 (US). WENT, Gregory, T.; 34 Scotland Avenue, Madison, CT 06443 (US).

(74) Agents: MORRIS, Francis, E. et al.; Pennie & Edmonds, 1155 Avenue of the Americas, New York, NY 10036 (US).

(81) Designated States: AL, AM, AU, AZ, BB, BG, BR, BY, CA, CN, CZ, EE, FI, GE, HU, IS, JP, KG, KP, KR, KZ, LK, LR, LS, LT, LV, MD, MG, MK, MN, MX, NO, NZ, PL, RO, RU, SG, SI, SK, TJ, TM, TR, TT, UA, UZ, VN, ARIPO patent (KE, LS, MW, SD, SZ, UG), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: APPARATUS AND METHOD FOR THE GENERATION, SEPARATION, DETECTION, AND RECOGNITION OF BIOPOLYMER FRAGMENTS



(57) Abstract

This invention is an integrated instrument for the high capacity electrophoretic analysis of biopolymer samples. It comprises a specialized high-voltage, electrophoretic module in which migration lanes are formed between a bottom plate (446) and a plurality of etched grooves in a top plate (438), the module permitting concurrent separation of 80 or more separate samples. In thermal contact with the bottom plate (446) is a thermal control module incorporating a plurality of Peltier heat transfer devices for the control of temperature and gradients in the electrophoretic medium. Fragments are detected by a transmission imaging spectrograph which simultaneously spatially focuses and spectrally resolves the detection region of all the migration lanes. The spectrograph comprises a transmission dispersion element and a CCD array to detect signals. Signal analysis comprises the steps of noise filtering, comparison in a configuration space with signal prototypes, and selection of the best prototype. Optionally post-processing is done by a Monte-Carlo simulated annealing algorithm to improve results. Optionally, an array of micro-reactors can be integrated into the instrument for the generation of sequencing reaction fragments directly from crude DNA samples.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia
AT	Austria
AU	Australia
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CS	Czechoslovakia
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgystan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakhstan
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam

APPARATUS AND METHOD FOR THE GENERATION, SEPARATION, DETECTION, AND RECOGNITION OF BIOPOLYMER FRAGMENTS

This specification includes an appendix containing a listing of the computer programs of this invention.

This invention was made with government support under grant numbers 1R43HG00960-01, 1R43HG01013-01A1, and 1R43CA65184-01 awarded by the National Institutes of Health. The government has certain rights in the invention.

FIELD OF THE INVENTION

This invention relates to a method and apparatus for analysis of biopolymers by the electrophoretic separation of biopolymer fragments. More particularly, it relates to a method and apparatus for automated, high-capacity, concurrent analysis of multiple DNA samples.

BACKGROUND OF THE INVENTION

Molecular biology research depends on biopolymer analysis. Conventionally, for this analysis, a biopolymer sample is first fragmented into shorter length biopolymer fragments by enzymatic or chemical means. The fragments are distinctively labeled with detection labels and then separated, often electrophoretically. The fragment pattern is then detected to obtain information about the structure and nature of the original biopolymer sample. These steps are typically performed separately with human intervention required to transfer the sample from one step to another.

A well known example of biopolymer analysis is DNA sequencing. See F. Sanger, et. al., DNA Sequencing with Chain Terminating Inhibitors, 74 Proc. Nat. Acad. Sci. USA 5463 (1977); Lloyd M. Smith, et. al., Fluorescence detection in automated DNA sequence analysis, 321 Nature 674 (1986); Lloyd M. Smith, The Future of DNA Sequencing, 262 Science 530 (1993), which are incorporated herein by reference. A prevalent sequencing method comprises the following steps. A DNA sample is first amplified, that is the DNA chains are made to identically replicate, usually by the polymerase chain reaction (PCR). From the amplified sample, nested sets of DNA fragments are produced by chain terminating polymerase reactions (Sanger reactions). Each chain fragment is labeled with one of four fluorescent dyes according to the chain terminating base (either ddATP, ddCTP, ddGTP, or ddTTP). These fragments are then separated according to their molecular size by polyacrylamide gel electrophoresis and the unique dyes detected by their fluorescence. The DNA base sequence can be simply reconstructed from the detected pattern of chain fragments.

Electrophoresis is the separation of molecules by differential molecular migration in an electric field. For biopolymers, this is ordinarily performed in a polymeric gel, such as agarose or polyacrylamide, whereby separation of biopolymers with similar electric charge densities, such as DNA and RNA, ultimately is a function of molecular weight. The prevalent configuration is to have the gel disposed as a sheet between two flat, parallel, rectangular glass plates. An electric field is established along the long axis of the rectangular configuration, and molecular migration is arranged to occur simultaneously on several paths, or lanes, parallel to the electric field.

DNA sequence information is key to much modern genetics research. The Human Genome Project seeks to sequence the entire human genome of roughly three billion bases by 2006. This sequencing goal is roughly two orders of magnitude (factor of 100) beyond the total, current yearly worldwide

DNA sequencing capacity. Sequ ncing of other biopolymers, for example RNA or proteins, is also crucial in other fields of biology. Other DNA fragment analysis techniques, such as PCR based diagnostics, genotyping (Ziegle, J. S. et al., Application of Automated DNA Sizing Technology for Genotypeing Microsatellite Loci. Genomics, 14, 1026-1031 (1992)) and expression analysis are increasing in use and importance.

The need for methods to identify genes which are differentially expressed in specific diseases such as cancer is of paramount importance, for both the diagnosis of the disease and for therapeutic intervention. Identification of genes specifically expressed in different diseases will lead to better classification of these diseases with regard to their biological behavior. A molecular understanding of disease progression is fundamental to an understanding of a specific disease. The identification of molecular diagnostics that correlate with variations in disease state, growth potential, malignant transformation and prognosis will have tremendous implication in clinical practice, including the diagnosis and treatment of the disease.

No current method adequately or efficiently addresses the need to identify, isolate, and clone disease-specific genes. A new biopolymer fragment analysis method has been developed based on the use of arbitrarily primed PCR (Williams, J. G., Kubelik, A.R., Livak, K. J., Rafalski, J.A., and Tingey, S.V., DNA polymorphisms amplified by arbitrary primers are useful as genetic markers. Nucleic Acids Res. 18, 6531-6535 (1990); Welsh, J. and McClelland M., Genomic fingerprinting using arbitrarily primed PCR and a matrix of pairwise combinations of primers. Nucleic Acids Res., 19, 5275-9 (1991)). When applied to mRNA, samples are first reverse transcribed into cDNA and then amplified with a combination of arbitrary and specific labelled primers (Froussard, P., A random-PCR method (rPCR) to construct whole cDNA library from low amounts of RNA. Nucleic Acids Res. 20, 2900 (1992); Welsh, J. et al., Arbitrarily primed PCR fingerprinting of RNA. Nucleic Acids Res., 20, 4965-70 (1992)). The resulting labeled DNA fragments are then electrophoresed through a gel producing a "banding pattern" or "fingerprint" of the mRNA source and run in separate gel lanes (Liang, P. and Pardee, A. B., Differential Display of Eukaryotic Messenger RNA by Means of the Polymerase Chain Reaction. Science, 257, 967-971 (1992)). Differences in gene expression are then found by manually comparing the fingerprints obtained from two mRNA sources. Following this, fragments of interest are extracted from the gel. This method is severely limited by its reliance on autoradiographic methods to allow for the isolation of the genes of interest. Refinements of PCR based techniques have, however, led to the ability to produce more reproducible banding patterns, and to the use of an automated DNA sequencing machine to record the banding patterns produced with fluorescently labeled primers (Liang, P., Averboukh, L. and Pardee A. B., Distribution and cloning of eukaryotic mRNAs by means of differential display: refinements and optimization. Nucleic Acids Res. 21, 3269-3275 (1993)). However, commercial automatic sequencing instruments (Applied Biosystems Inc., Foster City, CA, DNA sequencer) do not allow for the resolution of many dye labels or allow for the isolation of the fluorescently labeled samples after they are run. In an automated machine the sample is simply lost. Arbitrary primed PCR methods would be much more attractive if their limitations could be addressed.

To address these limitations, our invention allows these gene fragments to be detected fluorescently and to be directly isolated, without human intervention, as they are identified. This is accomplished by electrophoretically separating the individual bands, and hence the differentially expressed genes, from the rest of the sample as it is running. This approach incorporates the advantages of the PCR based methods to differential screening, while raising the level of speed, sensitivity and resolution well beyond that achievable with radiographic techniques. To insure high

Once fragment events are discriminated, the entire data for a run must be assembled to determine the nature of the original biopolymer sample. For DNA sequencing, this is conventional: the bases and their order in the DNA sample are the terminating bases of the fragments in the order of increasing molecular weight. When sequencing on a genomic scale, the bases and their order must be assembled into an ordered listing of the bases of the genome of the organism being studied.

All the foregoing technical requirements have prevented creation of an integrated machine for rapid, concurrent generation and analysis of large number of biopolymer fragment samples. The need for such a machine is widely felt in such areas as biological research, for example the Human Genome Project, the biotechnology industry and clinical diagnosis.

SUMMARY OF THE INVENTION

The apparatus and method of this invention have for their object the solution of these problems in electrophoretic biopolymer fragment analysis, and in particular, in DNA sequencing. In one aspect, the invention is an integrated, high capacity, low-cost machine for the automatic, concurrent analysis of numerous biopolymer fragment samples. Among its objects are the provision of: easily loaded, simultaneously observable, electrophoretic geometries comprising multiple migration lanes each of the order of 100 μm and down to 25 μm or smaller; a spectral detection system which is capable of sensitive, simultaneous response to signals emitted by all the migration lanes and which is dynamically adaptable, without physical intervention, to different dyes, different numbers of dyes, and different coding of fragments with dyes; automatic generation of multiple biopolymer fragments directly on the analysis machine from crudely purified biopolymer samples and bulk reagents (for DNA, sequencing reactions would be automatically carried out); and an automatic data analysis method for transforming time-series of spectral signal to biopolymer sequences and which is adapted to the unique problems of discriminating overlapping and weak fragment recognition events while achieving 99% or greater recognition accuracies.

A high capacity analysis machine according to this invention includes elements for concurrent loading of multiple samples for analysis onto the machine, an electrophoretic module for actually performing the sample separation, a spectrometer capable of simultaneous spatial and spectral resolution and detection of light signals representative of sample fragments as they are separated by the electrophoretic module, and elements for converting the detected signals into the sequence and character of the biopolymer samples analyzed.

Different sample loading techniques are used by different versions of this invention. One technique consists of simply loading small liquid volumes containing fragment samples -manually or automatically - into wells in the electrophoretic medium. More preferable is solid phase loading. Here a comb-like device has teeth which are sized and spaced to fit concurrently into all the sample wells in the electrophoretic medium. Each tooth carries a fragment sample attached by various denatureable binding methods. All the samples are released concurrently when the teeth are dipped into the sample wells. Advantageously, combs may have 50 to 100 teeth for concurrent loading of that number of samples. Notches machined in the comb insertion region can aid the sample loading by aligning the comb with the sample wells. Regardless of how the samples are loaded, the DNA fragments can be collected at a low voltage focusing electrode prior to the electrophoretic separation, thereby increasing the intensity and resolution of the analysis signals detected.

Most preferable, especially for DNA sequencing, is a reactor array to generate fragment samples from crude DNA and to inject them onto the electrophoretic module. The reactor separation resolution, it is advantageous for the gel throughout a migration lane to be kept as uniform as possible and for the lanes to be sufficiently separated to be clearly distinguishable.

To achieve these required improvements in the analysis capacity for DNA and for other biopolymers, machines are needed for the rapid, concurrent analysis of large numbers of minute biopolymer samples. Further, the analysis must be done with minimal human intervention and at low cost. Since electrophoresis will remain the dominant biological separation technology for the foreseeable near future, the technical demands of more rapid electrophoresis will shape the design of such machines.

More rapid electrophoresis requires, primarily, higher voltages and stronger electric fields to exert greater forces on migrating molecules and move them at greater velocities. However, higher fields and velocities lead to increased resistive heating and consequent thermal gradients in the gel. Gel non-uniformities result, impairing separation resolution. To preserve resolution, ever smaller gel geometries must be used so that this damaging heat may be more readily conducted away. Moreover, parallel, narrow migration lanes are advantageous to increase the number of samples analyzed simultaneously. While electrophoresis has been described in geometries where the parallel glass plates are spaced from 25 to 150 μm apart, instead of the usual 400 μm, it is not possible to insure long, parallel, narrow, and closely spaced migration lanes in such a thin sheet. Alternatively, electrophoresis has been described in arrays of capillary tubes down to 25 μm in diameter which completely define migration lanes. However, although the conventional plate arrangement is relatively easy to load with gel and samples, arrays of capillary tubes are much more difficult to load. Easy loading is advantageous to minimize analysis setup time and human intervention.

The small geometries required by high resolution, high voltage electrophor tic analysis create additional technical demands. Where fluorescent dye fragment labeling is used, sensitive spectral detection devices are needed. These detection devices must respond quickly, since rapid migration presents fragment samples for detection with only slight time separation. Most significantly, rapid parallel analysis of many biopolymer samples requires the detection device to simultaneously detect fragments migrating in separate lanes. Conventional detectors cannot meet these demands. One design uses rotatable filters to select spectral ranges to present to a single active detector element, this assembly being scanned mechanically across all the migration lanes. However, such mechanical single detector assemblies waste most of the available fluorescence energy from the fragment samples, limit detection speed, prohibit simultaneous detection, and slow sample analysis. Use of spectrally fixed filters also limits dynamic adaptation to different detection labels.

While a spatially compact disposition of the migration lanes might permit simultaneous observation, sample loading into the migration lanes prior to an analysis run requires physical access to the migration lanes. Access is easier and more rapid for widely spaced lanes. Conventional, flat-plate techniques have only straight, parallel lanes and cannot accommodate these divergent requirements.

A high throughput analysis machine would generate voluminous detection data representing the rapidly migrating biopolymer fragment samples. Manual analysis of such data is not feasible. To minimize human post analysis checking, these methods should achieve accuracies of 99% or greater. Further, the data would contain fragment detection events closely spaced, even overlapping, in time. Moreover, small electrophoretic geometries and small fragment sizes would generate only weak signals with increased noise. Prior electrophoretic devices, on the other hand, generated only clearly separated detection events with good signal intensities.

array comprises an array of micro-reactor chambers each with a minute inlet port and capillary inlet and outlet passages. The capillary passages are controlled by micro-machined valves. In one example a bubble, created by heating the capillary fluid, is used to control fluid flow through a capillary tube. The heating is by a resistive micro heating element formed by depositing a resistive thin film in the wall of the capillary. Leads are deposited to conduct current from an external controller to the heating element. To use this array, samples are introduced through the inlet ports; reagents are successively introduced through the capillary inlets; and fragment samples are ejected through the capillary outlets when reactions are complete. Reactions are facilitated by thermal control and heating elements located within each reactor.

Enabling the use of such a micro-reactor array for DNA sequencing is the use of dUTP rich PCR primers, a method of this invention. PCR amplification and Sanger sequencing can proceed sequentially without interference in one reactor by using the enzyme Uracil DNA Glycosylase (UDG). UDG digests dUTP rich PCR sequencing primers into fragments ineffective for initiating chain elongation in the subsequent Sanger sequencing reactions.

Also enabling the use of the microreactor array for DNA sequencing is the use of the enzymatic pretreatment of PCR products using a combination of Exonuclease I and shrimp alkaline phosphatase (United States Biochemicals, Cleveland, Ohio). The activity of both of these enzymes in PCR buffer eliminates the need for buffer exchanges. The Exonuclease I enzyme removes the residual PCR primers, while the shrimp alkaline phosphatase de-phosphorylates the dNTP's inactivating them. The removal of both the primers and excess dNTP's prevents them from interfering in the subsequent Sanger sequencing reactions.

Enabling the use of the microreactor array for other DNA fragment analysis methods including expression analysis, genotyping, forensics, and positional cloning is the direct incorporation of fluorescent labels onto the 5′ end of the original PCR primers. These primers can be either specific for known sequences, as in the case of genotyping or arbitrary as in the case of expression analysis. A series of different dyes can be used to allow the PCR amplification step to take place in a multiplex fashion within a single reactor.

Once the samples are loaded, separation occurs in the electrophoretic module. The invention is adaptable to use different such modules. One such module comprises rectangular plates spaced slightly apart to define a rectangular sheet of electrophoretic medium. Migration occurs in straight, parallel lanes through this medium. Another version uses ultra-thin plate spacing, down to 25 μm, and high electrophoresis voltages, thereby achieving rapid fragment separation.

The preferred electrophoretic module is constructed using two plates with a photolithographically generated formation of grooves bounded by the plates. Numerous non-intersecting grooves etched or otherwise formed on the top plate, together with the bottom plate, define the migration lanes. The lanes are therefore separate non-communicating channels for holding separation medium. Different groove and migration lane geometries are possible. One geometry is straight, parallel lanes. The preferred geometry spaces lanes widely at the loading end of the module, to ease the physical aspects of loading, but converges the lanes closely at the detection end, to permit simultaneous detection of separated fragments in all lanes. Groove size may be down to 25 μm to allow high voltage rapid electrophoresis. The grooves are preferably fabricated with standard photo-lithography techniques and, if necessary, subsequent etching and coating. Various combinations of substrates and processes are available including patterning insulators on conductive surfaces, patterning polymers on insulating/conductive surfaces, or patterning conductors and coating with insulators. Alternatively, a master mold can be formed photolithographically, followed by duplication of the grooves in disposable substrates via casting.

In all versions the highest allowable electrophoretic voltages are used, where the maximum voltage is determined as that at which the mobility of biopolymer fragments is no longer sufficiently length dependent. Thermal control is achieved with a thermal control module in good thermal contact with the bottom plate. The preferred electrophoresis module provides especially good thermal control, since the small separation medium channels are in close contact on all sides with top and bottom plates. The thermal control module has a heat sink adapted to heat exchange with an air or water exchange fluid. Between the heat sink and the bottom plate of the electrophoretic module are bi-directional heat transfer devices. Preferably these are Peltier thermoelectric modules disposed for pumping heat in both directions. Thereby, the bottom plate can be heated and cooled as needed and thermal gradients eliminated.

In one version, a transmission imaging spectrograph is used to detect separated fragments. The invention is particularly adapted to DNA sequence or other DNA analysis methods, in which each of the different fragment types is labelled with a different spectrally distinctive fluorescent dye. One or more lasers at the separation end of the electrophoresis module excites the dyes to emit light. Emitted light from samples in the migration lanes is incident on a collection lens. The light then passes first through a laser light filter, then through a transmission dispersion element, which spectrally separates the light, and finally through a focusing lens. The focused light is incident on a charge coupled device (CCD) array which detects the simultaneously spatially focused and spectrally diverged light from the detection regions of all the migration channels. Electronic signals from the CCD array provide information about the character or sequence of the DNA sample.

In the preferred version, a microfabricated set of components replaces the large scale imaging spectrograph. Here the function of the two camera lenses and diffraction grating is integrated within a single binary optic diffractive element. The diffractive element can be fabricated either on a glass surface, or on a separate material to be inserted between glass pieces.

The analysis system converts the electronic signals into biopolymer information which in one example is DNA base sequence. It comprises a standard programmable computer with short and long term memory and loaded with analysis programs particularly adapted to the preferred version of this invention. Interface devices place the electronic CCD output signals in the computer memory as binary signals. These signals are grouped both into spatial groups, one group for each migration lane, and into spectral groups, one group for each spectrally distinctive dye label. The grouped signals are filtered to minimize noise: high-pass filtering removes baseline low frequency noise, and low-pass filtering removes high-frequency single spike noise. If multiple samples are contained within a single migration lane, as enabled by the spectral multiplexing, the signals associated with each of the samples can be distinguished and grouped together using knowledge of the dyes associated with each of the samples.

The filtered signals are then compared to fragment recognition prototypes and the best prototype is chosen for each segment of filtered signals. The best prototype is that prototype whose averaged signal behavior for nearby times is closest to the observed signal behavior for the same nearby times. Closeness is simply measured by the ordinary distance between the observed signals and the prototypes. The base generating the input signals is identified as the base associated with the closest prototype. The sequence of closest prototypes thereby determines the DNA sequence and this sequence is output from the analysis system. In one embodiment, distances to each of the prototypes, or averages of the distances to the prototypes associated to the four possible bases, are also output. These values can be used to judge the confidence which one should assign to the DNA sequence, and in particular can be used to aid the comparison and assembly of multiple instances of the sequence of a given segment of DNA.

The prototypes are the averages of filtered signals generated in the apparatus of this invention from the analysis of known DNA. They are carefully chosen to be adapted to the characteristics of this invention. Preferably, they are chosen to include the signals generated by two sequential DNA fragments.

Further analysis is done in one embodiment of the invention. Any DNA sequences which are known (vector DNA) are trimmed out of the observed sequence. The remaining sequence is proofread by Monte Carlo simulated annealing. At random observation times a random alteration to the determined base sequence is made. The closeness between the entire resulting sequence and the entire filtered observed signal is evaluated. If a probabilistic test based on this closeness is met, the sequence alteration is retained; otherwise it is discarded. Alter and test activity is repeated until no further significant improvements occur. This step permits global improvements to be made in the overall sequence determined.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other features, aspects, and advantages of the present invention will become better understood by reference to the accompanying drawings, following description, and appended claims, where:

Fig. 1 shows an overall view of a preferred embodiment of the invention;

Fig. 2A shows details of the transmission imaging spectrograph that may be used in the device of Fig. 1;

Fig. 2B shows details of an alternative transmission imaging spectrograph of the device of Fig. 1;

Fig. 3 shows a ray trace of the transmission imaging spectrograph of Fig. 2;

Fig. 4 shows details of an alternative electrophoresis module for use in the device of Fig. 1;

Figs. 5A-5E show details of the process for making the electrophoresis module of Fig. 4;

Fig. 6 shows details of the module of Fig. 4;

Fig. 7 shows the operation of the module of Fig. 4;

Fig. 8 shows details of an array of two micro-fabricated reactors of the device of Fig. 1;

Figs. 9A-9B show a valve design for the reactors of Fig. 8;

Figs. 10A-10D show the generation of solid phase fragments in the device of Fig. 1;

Fig. 11 shows the steps of a dUTP digestion process;

Fig. 12 shows an overall flow chart of the analysis steps used in practicing the invention;

Fig. 13 shows the flow chart for the analysis preprocessor step of Fig. 12;

Fig. 14 shows the general operation of the basecalling step of Fig. 12;

Fig. 15 shows the flow chart for the analysis basecalling step of Fig. 12;

Fig. 16 shows the flow chart for the analysis proofreading step of Fig. 12;

Fig. 17 shows a recording of an illustrative output of the invention;

Figs. 18A, 18B and 18C show recordings of illustrative output of the invention from three separation runs; and

Fig 19A and 19B show recordings of the output of the spectrograph of Fig. 2A.

DETAILED DESCRIPTION

Instrument Overview

Fig. 1 illustrates a preferred embodiment of the integrated biopolymer analysis instrument of the invention. Only essential components are depicted; non-essential mechanical components conventional in instrument design are not depicted. The following is a general description of the instrument and its use. Detailed descriptions of the construction and use of components follow.

Element 104 is an electrophoresis module. As illustrated, it comprises a micro-fabricated gel electrophoresis plate (microFGE) 106, a micro-fabricated reactor array (microFRA) 110, and a temperature control subunit 108. MicroFGE 106 comprises converging electrophoresis migration lanes 107 formed as grooves in a glass or plastic plate and containing separation medium. Biopolymer fragments differentially migrate in these lanes from left to right under the influence of an electric field supplied by driving electrodes (not shown) at opposite ends of the electrophoresis module. In other versions, the microFGE could have lanes of other geometries, for example, parallel lanes. It could also be replaced with a conventional non-grooved glass plate. MicroFRA 110 is the source of samples of biopolymer fragments for analysis. The samples are generated from raw biopolymer samples in the micro-reactors of the array and loaded directly into the electrophoresis plate typically with a different sample in each migration lane, or with multiple samples contained in a given migration lane. Illustratively, the fragments are labelled with one of four fluorescent dyes according to the chain terminating base (either ddATP, ddCTP, ddGTP or ddTTP) as is known in the art. In other versions, the microFRA could be replaced with a solid or liquid phase loading apparatus.

At the right are one or more lasers 102 that generate a collimated beam 113 that is directed to pass transversely through the microFGE in an unobstructed laser channel 115. The terminal ends of the migration lanes 107 intersect this channel. The beam thereby simultaneously illuminates the separated biopolymer fragments in the different migration lanes and excites their labels to fluoresce. A transmission imaging spectrograph 100 is disposed above the beam. The spectrograph has within its field of view all the converged migration lanes in microFGE 106 and is equipped to make simultaneous spectral observations of fluorescence in all of the migration lanes. Light resolved by the spectrograph is converted into electronic signals representative of the different fluorescent labels that are excited. As a result, the separated biopolymer fragments are detected.

Electronic signals representing these observations are read into a controller/power supply 114 for on-line or off-line processing by a computer 112. The computer performs an analysis adapted to the characteristics of an individual biopolymer analysis instrument and its particular running conditions. The analysis method generates information characterizing the original biopolymer samples, for example DNA base sequences or genotypical character.

Optionally, the computer can also control an analysis run by commanding the controller/power supply to generate necessary voltage outputs. For example, controller/power supply 114 generates the high voltages applied through leads 116 to the driving electrodes to drive molecular migration in the electrophoresis module. If the microFGE has the optional capability to shunt fragment samples between migration lanes as described below in conjunction with Fig. 7, the controller/power supply also generates necessary shunting voltages which are applied to shunting electrodes 118 in the microFGE module. Other voltages which are supplied by the controller/power supply 114 include those which cause the loaded DNA fragments to concentrate at a focusing electrode prior to the initiation of electrophoresis.

Transmission Imaging Spectrograph

The transmission imaging spectrograph 100 is designed to resolve spectra within the range of common dye labels used in biopolymer analysis (approximately 500 nm to 700 nm), to have high light gathering ability, and to have a wide field of view with little light loss for peripheral images. These features permit the simultaneous viewing of many migration lan s. Advantageously, spectrograph 100 may have a spectral range on the order of 400 nm to 800 nm. Fig. 2A illustrates one version of this component. Non-essential mechanical components conventional in instrument design are not depicted.

As indicated previously, laser 102 generates laser beam 113 which is directed through laser channel 115 so as to intersect electrophoresis migration lanes 107. Light is scattered from this beam primarily by two mechanisms. First, there is some scattering at the laser wavelength by the separation medium and other matter traversed by the beam. Second, when a labeled fragment passes through the beam, it is excited and fluorescence at characteristic wavelength(s) is emitted in all directions.

A portion 240 of this scattered light is incident on spectrograph 100. Spectrograph 100 comprises a collection lens 222, a laser rejection filter 236, a transmission dispersion element 224, a focusing lens 226 and a charge coupled device (CCD) array detector 228. The CCD array comprises a two-dimensional array of CCD detector elements oriented with its short axis along spectral divergence axis 244 and its long axis along spatial focusing axis 245. This orientation gives adequate spectral range and maximal spatial range. Electronic data output from the CCD is transferred to the controller/power supply.

Collection lens 222 collimates the scattered light into parallel rays. Collimated light then passes through laser rejection filter 236, which absorbs light at the laser wavelength. The remaining filtered light, which consists essentially of fluorescence from the fragments, then passes through transmission dispersion element 224, which can be either a grating prism (known as a grism), as illustrated, or alternatively a transmission diffraction grating. This element separates the light into rays of differing wavelength, which diverge along the direction of spectral axis 244. Focusing lens 226 then focuses the light on CCD array detector 228.

Images of the fluorescing fragments in the different lanes 107 are formed along spatial axis 245 and simultaneously separated by wavelength along spectral axis 244. In this manner, different dye labels in different migration lanes produce different patterns along the spectral and spatial axes and can be simultaneously discriminated.

Fig. 3 illustrates the optics of spectrograph 100, spectral dispersion being out of the plane of the diagram. To maximize the field of view focused on the detector and to minimize loss of light at the edges of the field of view, distance 330, between the collection lens 222 and the focusing lens 226, should be as short as possible. As a result, only extreme off-axis rays such as ray 334 will completely miss detector 228 and the optical parameters of the spectrograph can be selected so that sufficient fluorescence from each of the migration lanes is incident on detector 228 to permit identification of the labelled bases. To achieve minimum distance 330, the wavelength dispersive element is preferably a transmission dispersion element, either a transmission grating or a grism.

The following components are exemplary for one version of the transmission imaging spectrograph. For collection lens 222, a Pentax 165 mm f2.8 lens or a 250 mm f5.6 Sonnar medium format camera lens from Carl Zeiss is used. These lenses are commercial camera lenses for use with medium format photography chosen both for their large numerical aperture and wide field of coverage and to match the demagnification required by the other components in the system. Preferably, the lens is a 400 mm f8 Osaka large format lens, or other large format long focal length lens which enables simultaneous imaging of a wide cross-section of electrophoresis module 104. Laser rejection filter 236 is a Raman Edge Filter REF521 from Omega Optical Inc. (Brattleboro, VT). It has an optical density of 3 to 4 at 515 nm, a transmission greater then 80% over most of the design spectral range, high absorption near the laser wavelength, and behaves well for light incidence off the optical axis. Preferably, laser rejection filter 236 is one or several holographic notch rejection filters from Kaiser Optical Systems (Ann Arbor, MI) which enable the use of multiple laser excitation sources by transmitting light of both lower and higher energy than that of the rejected laser light. Transmission dispersion element 224 is a Diffraction Products (Woodstock, IL), 3090-84ST transmission grating with: 600 grooves/mm, a large clear aperture of 84 mm x 84 mm, a back face single layer MgF_2 anti-reflection coating, and best efficiency at 500 nm with a first order grating efficiency of approximately 50%. Focusing lens 226 is a Canon 85 mm f1.2 lens. This is a commercially available 35 mm format camera lens with aspherical elements and special low dispersion glass, allowing the design to be optimized for a very large numerical aperture. The CCD array detector 228 is a Princeton Instruments Inc. (Trenton, NJ) TE/CCD 1024E Detector with a ST 130 DMA Controller. This array detector is 1024 x 256 pixels with pixel size 27 x 27 μm and operated in multi-pinned phase mode with fast readout along the long axis. This Grade 1 CCD has a large physical dimension along the long axis which provides the spectrograph with a wide field of spatial coverage (the entire width of the gel) while limiting the demagnification required by the lenses selected. Alternatively, a frame transfer CCD can be used that allows for transfer of an image rapidly to a masked portion of the array for subsequent readout, providing a very rapid rate of sequential image acquisition. Lasers 102 can be, for example, Argon ion or solid state or HeNe, with exemplary single laser wavelengths being 514.5 nm or 488 nm (Argon ion) or 532 nm (NdYAG solid state) or 523 nm (NdYLF solid state) or 633 nm (HeNe), and exemplary pairs of laser wavelengths being 514.5 and 633 nm, or 532 and 633 nm. A direct doubled solid state diode laser (Coherent Inc., Santa Clara, CA) at 430 nm can be used in a triple laser excitation instrument together with 532 nm and 633nm lasers and appropriate holographic notch rejection filters to achieve the ability to simultaneously excite and detect all dyes in the wavelength range 440 to 700 nm. Illustratively, in the case of two laser wavelengths at 514.5 nm and 633 nm, and five dyes, the green laser is used to excite the FAM, JOE, TAMRA, and ROX dyes (Applied Biosystems/Perkin Elmer, Foster City, CA) and the red laser is used to simultaneously excite the Cy5 dye (Biological Detection Systems, Pittsburgh, PA). By way of further example, the FAM, JOE, TAMRA, and ROX dyes are used to label the ddCTP, ddATP, ddGTP, and ddTTP reactions from the forward primer, and the Cy5 dye is used to label one reaction, say ddTTP, from the reverse primer. All five dyes are recognized while being run simultaneously in a single migration lane.

One version of the imaging spectrograph was designed for a spectral range of approximately 510 nm to 640 nm, which spans the fluorescence wavelengths of many dye labels. The one-dimensional grating equation is:

$$n\sin(\alpha)-\sin(\beta) = \frac{m\lambda}{\sigma} \qquad (1)$$

where m is the order number, λ is the wavelength (in nm), σ is the groove spacing, n is the index of refraction of the grating material, and α and β are the angles of incidence and diffraction, respectively. For first order (m = -1), 600 groove/mm grating, and 0° incidence angle:

$$\sin(\beta) = \frac{\lambda}{1667} \qquad (2)$$

Thus 510 nm light diffracts at an angle 17.8°; 575 nm light at 20.2°; and 640 nm light diffracts at 22.6°. With an 85 mm focal length second lens focused at infinity, and 575 nm light directed to the center of the short axis of the CCD camera, then either 510 nm or 640 nm light (diffracted by 2.4 degrees less or more than 575 nm light, respectively) will strike the CCD array at a distance y in mm where

$$\tan(2.4) = \frac{y}{85mm} \qquad (3)$$

Computing, y = 3.56 mm. This corresponds to 132 pixels in the CCD camera with 27 μm per pixel, just slightly more than the 128 pixels available from center to edge of the short axis.

Thus these components provide a version of the spectrograph with adequate spectral resolution over the spectral design range. If desired, CCD array 228 can be rotated by 90° enabling observation of fluorescence over an extended spectral range from 500 nm to near infrared, but over a reduced spatial range. Optionally, a grating with lower groove density (300 grooves/mm) can be used to increase the spectral range observed while maintaining spatial coverage.

Alternatively, the above components can be reduced in size and integrated into a microfabricated imaging spectrograph positioned in contact with a CCD array. A cross-section through one of the many channels of a binary-optic spectrograph array is shown in Fig. 2B. Here the two camera lenses and diffraction grating of Fig. 2A are replaced by a single binary diffractive element 237 located between supporting glass elements 238, 239. This diffractive element can be fabricated on a glass surface as shown or separately on a material to be inserted between glass pieces by conventional photo-lithograph techniques. The fabrication of similar microlenses is known in the art. See, for example, W.B. Veldkamp et al., "Binary Optics," Scientific American, 266:5, pp. 92-97 (1992) which is incorporated herein by reference. To form the binary diffractive element, SiO2 is typically deposited onto a glass surface and is then patterned using standard e-beam techniques.

Electrophoresis Module

The electrophoresis module is designed to provide a maximum number of small, closely spaced migration lanes, to allow use of high voltages, to dissipate resistive heat, to maintain high resolution, and to be adaptable to alternative sample loading means. Together with the transmission imaging spectrograph, these features promote rapid, concurrent analysis of many biopolymer samples. Fig. 4 illustrates the electrophoresis module. In this figure microFRA 110 has been replaced by a solid phase loading means. Alternatively, a conventional liquid phase loading means may be used. Only essential elements are depicted. Elements conventional in instrument design are omitted. See, for example, US Patent 5,228,971, Brumley et. al., Horizontal Gel Electrophoresis Apparatus (Jul. 20, 1993); US Patent 5,137,613, Brumley et. al., Horizontal Gel Electrophoresis Apparatus (Aug. 11, 1992); and US Patent 5,171,534, Smith et. al., Automated DNA Sequencing Technique (Dec. 15, 1992) which are incorporated herein by reference.

The electrophoresis module comprises a top plate 438, a bottom plate 446, end pieces 458 and 459 and a comb pressure piece 456. Bottom plate 446 provides support and attachment for other module components and serves as the bottom of the migration lanes and buffer wells. Component attachment can be with conventional thumbscrew clamps or other standard mechanical devices. Positioned and attached at the left and right ends of the bottom plate are two end pieces 458 and 459. The end pieces include electrodes for applying high voltage across the migration lanes. The end pieces have a substantially "U" shape, defining buffer wells 442 within the arms of the "U". Buffer solution in these wells is in contact with the separation medium in the migration lanes. The end pieces are sealed to adjacent elements by elastomer seals 454. Left end piece 458 is sealed to comb pressure piece 456, and right end piece 459 is sealed to the right end of top plate 438.

Positioned, attached, and sealed adjacent to the left end piece is comb pressure piece 456. The pressure piece permits liquid communication between the left buffer well and the separation medium. Between the pressure piece and top plate 438 is gap 463 which guides the insertion of a well-forming comb and, optionally, a solid phase loading comb. One such comb is shown in Fig. 4 having a base 460 and numerous teeth 462. The other comb is similar except as noted below. The well-forming comb is used in a conventional manner to form sample loading wells in the separation medium in gap 463. Prior to polymerization of the separation medium, this comb is inserted in gap 463 and fixed in position by a horizontally applied force between the comb pressure piece and top plate 438. This force is conventionally generated by adjustable attachments bearing horizontally against the left end piece 458 so as to bias the pressure piece against the comb. Once the separation medium has polymerized, the well-forming comb is removed leaving sample loading wells at the position of the teeth. In a preferred embodiment, comb pressure piece has machined notches 461 that match the comb teeth 462 to provide rigid formation of wells and aid sample loading. In another embodiment, a "shark's tooth" comb is used instead of the well forming comb. As is known in the art, the shark's tooth comb has one substantially flat edge containing small protrusions, and an opposite edge containing multiple teeth. The comb is first inserted into gap 463 with the substantially flat edge oriented toward the bottom plate, in order to form a thin layer of separation medium along the bottom of the gap. The separation medium is then polymerized and the comb is withdrawn. The comb is then inverted and reinserted into gap 463 so that the teeth impinge upon the thin layer of separation medium. Wells are formed in the spaces between the teeth when the teeth are compressed between the top plate 438 and comb pressure piece 456. Samples are then loaded into the wells formed between the teeth of the comb.

A solid phase loading comb may also be guided into gap 463 to load biopolymer fragment samples prior to analysis. The teeth of the loading comb are spaced and sized to fit in the sample wells formed by the well-forming comb and, in the case of the preferred embodiment, in the notches machined in comb pressure piece. The teeth have the same center-to-center spacing as those of the well-forming comb but are smaller in size. Fragment samples are bonded to the teeth of the loading comb, the comb is guided by notches 461 into gap 463 so that the teeth enter the sample wells, and the fragment samples are released into the wells. The technique achieves rapid, error free, parallel loading of all the samples for analysis. For further details concerning parallel sample loading, see A. Lagerkvist et al., "Manifold Sequencing: Efficient Processing of Large Sets of Sequencing Reactions," 91 Proc. Nat. Acad. Sci. USA, 2245 (1994) which is incorporated herein by reference.

Alternatively, conventional liquid phase loading may be used. In such case, small liquid volumes containing the fragment samples are directly placed into the sample wells. Various conventional mechanical devices may be employed to speed up and reduce errors in this manual process.

After injection of samples into the loading wells, but prior to electrophoretic separation, the DNA samples can be concentrated using a focusing electrode. Low voltage applied to the focusing electrode, located within the migration path near the load well, causes the DNA samples to migrate and gather at the electrode. Immediately following such focusing, electrophoretic separation is initiated. This avoids any broadening associated with diffuse entry of samples into the separation medium. As a result, the detection of signal temporal resolution is maximized. Furthermore, since the samples are concentrated into small regions of the migration lane and since substantially all the DNA fragments loaded enter into the separation medium, signal intensity is maximized.

The focusing electrode is required to adhere well to the top plate 438 or bottom plate 446, and should limit the electrolytic generation of gas bubbles during the collection of the loaded DNA fragments. It can be fabricated onto a typical substrate using the following process, known in the art.

The process begins with depositing a thin (~100 Angstrom) adhesion layer of titanium metal onto the substrate, and then depositing a layer of platinum around 1000 Angstrom thick on top of this adhesion layer. These depositions can be done by thermal evaporation, electron beam evaporation, sputtering or any other technique that yields a clean, uniform layer. Next, photoresist is spin coated on top of the metal and patterned using standard photolithographic processing. The pattern produced in the photoresist is transferred to the underlying metals by etching. This can be accomplished by a liquid chemical etch, by a dry plasma etch, by a photoresist lift-off procedure or by any other method which accurately reproduces the pattern of the photoresist. Once this etching is completed, the photoresist is removed and the substrate cleaned to leave the final electrode pattern.

The microFGE top plate, illustrated in Fig. 4, includes numerous similar etched migration lane grooves 107 of roughly semi-circular cross section and of diameter between 10 and several 100 μm. The microFGE is positioned and attached in close contact with the bottom plate so that the etched grooves form individual, isolated migration lanes. The lanes are bounded on the bottom by the bottom plate and on the top and sides by semi-circular microFGE grooves. In Figs. 1 and 4 the etched grooves are illustrated as straight and converging at the laser illumination and detection region. Alternative lane geometries are possible. A preferred geometry includes grooves with first, straight sections that are widely spaced communicating with second sections that converge to a narrow spacing.

Instead of being etched with grooves, top plate 438 may be a conventional glass plate such as a sheet of optical quality glass, such as BK-7, polished to within 1 μm flatness. Such a sheet would be separated 25 to 150 μm from bottom plate 446 by polyester spacer gaskets.

Laser channel 115 is formed from an etched laser groove 457 extending across the plate with a depth not less than that of each of the migration lane grooves. Laser windows 444 cover the ends of the laser groove.

As shown in Fig. 4, a laser beam 113 from laser 102 is directed through channel 115 and illuminates fragments migrating down all the lanes. Alternatively, the laser can be brought into the lane first by directing it through the top or bottom plate and then by causing it to reflect from a suitably positioned mirror mounted within channel 115 so that it propagates through the laser channel. As still another alternative, individual laser sources can be fabricated into each lane by means of known photolithographic processes. In another mode of the invention, the laser can be caused to fan-out into a narrow sheet using line-generating optics which are known in the art. The sheet of laser light is directed to pass through the upper or lower glass plate, into and across the laser channel, where it can be focused to a line, thereby illuminating the fragments migrating down all the lanes.

Prior to an analysis run, a separation medium 451 is placed in all migration lane grooves 107 and laser groove 457 to resolve the fragment patterns. Separation medium 451 within the grooves is in contact with liquid buffer in buffer end wells 442. Most separation involves the use of polymer sieving media, either cross-linked gels or linear liquids. Most are based upon polyacrylimide. For example, when unpolymerized polyacrylimide is introduced into the lanes as a liquid, it polymerizes over a few minutes. Rarely is the media reusable and the careful cleaning required is labor intensive.

Alternative separation media are possible with these systems. Recent work has shown that 0.5 micron posts of SiO2 can retard the mobility of like sized DNA fragments (10 kilobases) to enable size sieving. W.D. Volkmuth et al. "DNA

Electrophoresis in Microlithographic Arrays," Nature, 358, 600 (1992). Reducing the dimension of these posts to the 50 nm size will increase the resolution to nearly base-pair. Another alternative may be offered simply by employing solid polystyrene spheres of an appropriate size. Huber et. al, "High-resolution Liquid Chromatography of DNA Fragments on Non-porous Poly(Styrene-Divinylbenzene) Particles," Nucleic Acids Res., 21, 1061-1066 (1993).

Fig. 6 illustrates temperature control subunit 108 and bottom plate 446. Bottom plate 446 may be made from a single material, such as glass or sapphire. Preferably it comprises a top plate or coating 676 of a chemically and electrically resistant material, such as a glass, silica, or diamond-like-carbon in substantial contact with a bottom plate 648 made of a highly heat conducting material, such as copper or aluminum. The heat conducting bottom plate may contain conventional water channels or air fins for efficient heat transfer with circulating water or air.

Preferably, the bottom plate is in contact with temperature control subunit 108. This subunit enables precise control of the separation medium temperature and ensures its uniformity. The elimination of injurious separation medium temperature gradients is vital to good electrophoretic resolution. The subunit comprises a heat sink 652 for transferring heat. The heat sink may contain water channels or cooling fins for efficient heat transfer with circulating air or water. A number of Peltier-effect thermoelectric heat pump assemblies 650 are mounted in good thermal contact between heat sink 652 and bottom plate 446. These heat pumps are mounted for rapid bi-directional heat transfer between the bottom plate, and thereby the separation medium, and the heat sink. They are powered by controller/power supply 114 in response to temperature input from thermocouple(s) 678 in contact with the bottom plate. As a result, bottom plate 446 is maintained at a desired, uniform, operating temperature, which may range from ambient to 90° C. The top plate can also be controlled in a similar manner.

Electrophoresis Module: MicroFGE

An industry standard, photolithographic fabrication process is used to fabricate the migration lane grooves and laser groove in the microFGE. A photolithographic mask with an etching pattern is constructed in a standard manner. Two patterns have been used. One has 80 straight, parallel, 11 cm long, 300 μm wide grooves spaced on 1.125 mm centers. The other has, at the left, 80 straight, parallel, 2 cm long, 50 μm wide grooves spaced on 1.125 mm centers, in the middle an angular bend, and at the right, straight, approximately 4 cm long, 50 μm wide grooves converging to 300 μm spacing. At the extreme right of both geometries is a 5 mm wide groove across the plate for the laser channel. Because the etch solution undercuts the etch mask during etching, the actual photolithographic groove width is less than the desired microFGE groove width.

The photolithographic mask is used in a conventional etching process comprising the steps illustrated in Figs. 5A-5E. The substrate is a 12.7 cm x 12.7 cm glass plate 166 polished to less than 1 μm flatness on both sides. Both plate surfaces are first prepared with a standard ammonia/hydrogen peroxide RCA clean. As shown in Fig. 5B, a silicon carbide (SiC) etch mask 168 is deposited using Plasma-Enhanced Chemical Vapor Deposition (PECVD). The reactants are CH_4 and SiH_4 with flow rates of 65 and 12 sccm, respectively; the power is 50 mW/cm^2; and the substrate temperature is 250°C. A five minute deposition on both sides of the glass plate leaves a thin SiC layer. This is followed by a dehydration bake for good photoresist mask adhesion.

Photoresist is then spin-coated on the front and a similar protective material is applied to the back of the glass plate. Illustratively, the spin-coating apparatus is a Shipley 1813 operating at 4000 RPM. Next, the photoresist is exposed through the photolithographic mask to a total exposure of 200 mJ/cm^2 at 405 nm and then developed. As shown in Fig. 5C, the photoresist pattern is transferred to the SiC etch mask using CF_4 reactive ion etching (RIE) at 40 mT and 300 mW/cm^2 for 7 min. to form a patterned etch mask 170. Over etching insures that no residual SiC remains in the exposed areas. Remaining photoresist is not stripped, as it serves to plug holes and defects in the SiC etch mask. As shown in Fig. 5D, the plate is then etched by immersion in a buffered HF (6:1) etch solution. The solution is stirred. The average etch rate is approximately 0.55 μm/min. A total etch time of 150 minutes yields 75-85 μm grooves. After etching, the plates are rinsed in a second bath of HF, then in H_2O. The remaining resist and back-side protective material are stripped (Shipley 1165) and the SiC is removed by RIE in a similar process to that used in defining the patterned SiC etch mask 110. As a result, this process transfers the pattern in the photolithographic mask to a pattern of approximately semi-circular grooves 174 on substrate plate 166.

In a preferred mode of the invention, Borofloat glass (Schott Corporation, New York) is used as the substrate, and the following masking and etching protocol is employed to generate grooves with smooth etch interfaces. The substrate is first cleaned using a standard RCA or piranha method known in the art. A thin layer (~100 Angstrom) of chromium is deposited on the substrate using high temperature evaporation. This chromium acts as an adhesion layer for a gold layer which is deposited on top. The gold layer is also formed by using high temperature evaporation, and has a thickness of around 1000 Angstrom. Next, photoresist is spun coated on the substrate and patterned with a series of lanes from a photomask using standard lithography methods. The pattern in the photoresist is then transferred into the gold and chromium layers using wet etching. Now, the etching of the lane pattern into the glass substrate begins. The etching is done using a solution of hydrofluoric acid (HF) in water. Typically, 49 wt% HF is used in order to minimize the etch times for deep channels. The glass etch rate with 49 wt% HF is around 7 micron/min for channels up to 100 micron deep. The advantage of the Borofloat is the absence of insoluble reaction products formed during etching which lead to rough etching of the side walls and bottom of the channels. No insoluble etch products are formed, and very quick etching and extremely smooth channel side walls and bottoms are formed. This smoothness is essential for the separation of biopolymer fragments because surface roughness can lead to abnormal mobility of DNA fragments during electrophoresis due to interactions of the fragments with the walls.

In another version of the invention, it is desired to achieve etched migration lanes in long channels (> 12 inches). In this case, the typical lithographic tools for microelectronics fabrication cannot be used because they are limited to circular geometries up to around 12 inches. In order to process these larger substrates, different tools must be employed to deposit and pattern the photoresist for lane formation. For the photoresist deposition, three primary methods exist: dip coating, spray coating and spin coating. In dip coating, the large substrate is dipped into a photoresist solution and slowly withdrawn to leave a uniform layer of photoresist coating the surface. In spray coating, photoresist diluted in solvent is air sprayed onto the substrate to coat the surface. To do spin coating, one employs a machine capable of processing very large substrates such as those used to fabricate large flat panel displays. Exposing the lane pattern onto the photoresist can be done in two ways - with a very large photomask or with controlled exposure such as through a laser writing system or a controllable aperture on a light source. Once the photoresist has been exposed, all the process steps detailed above can be carried out to form lanes in the glass substrate.

There are a number of variations on the above method, including both substrate and processing. If an insulating layer on a metal platform is sufficiently thick, then the microFGE pattern may be etched directly into this layer with RIE, and the metal will remain flat (unpatterned). A number of thin insulators can be used. A glass or quartz wafer can be bonded or epoxied to the substrate. Alternatively, a number of standard polymers used in the microfabrication industry include polymethylmethacrylate (PMMA) and polyimide (PI). These can be spin-coated onto flat, rigid heat-conducting substrates such as silicon, copper or aluminum. RIE patterning of polymers is typically done with a very simple chemistry such as O_2, and etch rates can be very high. The etched polymer microFGE is coated with PECVD Si or SiO_2 to facilitate polymerization of the separation medium. Optionally, a conducting substrate can be patterned directly followed by coating with a thin insulating layer. One can prepare standard microFGE plates etched in silicon similar to existing examples in glass, and coat them both with PECVD SiO_2, and PECVD diamond-like-carbon (Diamonex, Inc., Pennsylvania). The coatings will be in the range of 5-20 microns thick.

In an alternative embodiment of the invention, the photolithographic process can be used to generate a mold, against which multiple microFGE plates can be cast. The associated reduction in processing cost enables the microFGE plates to be disposable, easing the cleaning burden and cost associated with the DNA analysis process.

In more detail, any of the following protocols can be used for generating a pattern of lanes in a polymeric substrate.

(1) Direct patterning: The first method is direct patterning of lanes in a photopatternable polymer layer, with a specific example being a photopatternable polyimide, OGC Probimide 7020 (OCG Microelectronics, New Jersey). The precursor polymer can be applied from solution using standard spin coating techniques and directly exposed with UV light using standard photolithographic tools and a lane geometry photomask. The exposed areas of the polyimide are partially

cured by the UV light, and the remaining areas can be selectively removed with a suitable solvent. Finally, the polyimide is fully cured with heat to obtain a solid polymer. The top plate formed by the above (or another) approach must be bonded to a bottom plate to enclose and seal the channels. A thin polymer layer is spin coated on a glass bottom plate, and cured after contacting the top substrate. In this manner, the polymer film becomes an adhesive fusing the top and bottom. Alternatively, a bottom plate with a thin elastomeric layer can be reversibly sealed to the top plate with pressure.

(2) In-Situ pattern formation: The second protocol is in-situ curing of channels in a polymer layer between two glass plates This begins with a mixture of reactive monomers and oligomers in the liquid state (e.g. methyl methacrylate monomer and oligomer). This mixture is introduced between two glass plates spaced by the desired channel height. A photomask is then used to expose selected areas of the liquid to UV light to define the lanes. The areas exposed to the UV light react to form solid polymer walls defining the lanes. The unreacted liquid in the channels can then be pumped out to leave the final lane geometry. In this method, both the top and bottom glass surfaces can be coated with the polymer prior to channel formation to provide a uniform channel surface.

(3) Replication: The third protocol involves replication of the surface features of a microfabricated master in plastic using molding or casting techniques. Silicon, silica, metal or polymeric masters are fabricated with a negative of the desired pattern. A polymer substrate is solidified while in contact with this master to form the channels. Both molding (solidification due to cooling) and casting (solidification due to polymerization and crosslinking) can be used. Casting can be done at room temperature and pressure and is gentler on the master than molding. After forming the top plate, the bottom plate is bonded by the same technique described above. The mastering concept of the replication protocol can be used

to dramatically reduce the cost of microfabricated lane substrates when compared to etching the lanes in a glass substrate, enabling the use of disposable microFGE plates.

Electrophoresis Module: microFGE Shunting Capability

Micro-fabrication techniques permit the microFGE to be optionally configured with the capability to automatically collect selected biopolymer fragment samples. Samples traveling down a sample migration lane are detected in the laser illumination and detection region and analyzed by the computer implemented analysis method. If a particular sample is of interest, the computer can command that it be shunted into an adjacent, empty collection lane. After the analysis run is complete, the shunted sample can be further analyzed in or eluted from the collection lane.

Fig. 7 illustrates two adjacent lanes in the microFGE configured for this biopolymer fragment shunting capability. Adjacent migration lanes 762 and 764 are bounded at one end by end piece 458 and at the other end by driving electrodes 117. Downstream of the laser illumination region is a shunting cross-lane connector 784 with shunting electrodes 118 placed in the walls of the adjacent lanes. The shunting electrodes are connected to and controlled by controller/power supply 114. The cross-lane connector is fabricated by altering the photolithography mask to define the additional etching of the connector. During an analysis run, it contains separation medium. The shunting electrodes are placed by a conventional metallic deposition process.

During an analysis run, biopolymer fragments 780 migrate down the sample lane 764. The fluorescent emission of each fragment is detected as it crosses laser beam 113 and is analyzed by the analysis system. If a biopolymer fragment 782 is determined to be of interest, it is shunted from its sample lane to adjacent collection lane 762 by applying a voltage across shunting electrodes 118 when the sample is adjacent to the cross-lane connector. Sufficient voltage is applied for a sufficient duration to cause migration into the collection lane. The computer commands the controller/power supply to apply voltage at the correct time for the correct duration.

Fragment Generation and Loading: Liquid and Solid Phase Methods

The instrument of this invention is adaptable to a number of generation and loading methods for the biopolymer fragment samples. Liquid phase loading is conventional. This merely requires transferring liquid samples containing biopolymer fragments to sample wells. This is usually a sequential, slow, error prone manual step. Various mechanical and fluid devices may improve speed and error rate. However, it would be advantageous to load multiple samples in one operation in parallel. Even more advantageous would be the capability to generate fragment samples from raw biopolymers and to load them in parallel for analysis.

Parallel DNA fragment sample loading can be accomplished by solid phase loading, which is also conventional, although recent, technology. It is illustrated in Figs. 10A -10D. See Lagerkvist, et. al. (1994). As shown in Figs. 10A and 10B, a solid-phase loading comb 1006 has 48 or more teeth 1007, advantageously at least 96 teeth. The tips of the teeth are coated with streptavidin. The comb and teeth are designed so that the center-to-center tooth spacing matches the center-to-center spacing of the loading well of the microFGE and all the loading comb teeth can all be inserted into all the loading wells simultaneously. In a preferred method shown in Fig. 4, the teeth of the comb match the spacing of notches 461 machined in comb pressure piece 456 and form robust, mechanically strong, sample wells.

The comb is loaded with biotinylated PCR product samples generated from biotinylated PCR primers. The biotinylated samples are bound to streptavidin coated teeth 1007 by the strong streptavidin/biotin attraction. For example, up to 100,000 copies of a single DNA sample bind to each tooth. Standard Sanger sequencing reactions are then performed on the samples attached to each tooth of the comb to generate

DNA fragments 1010 shown in Fig. 10C. The geometry of the PCR and Sanger reactors is advantageously adapted to the comb spacing, so that the comb may be successively dipped in reactors with the appropriate reagents for performing these reactions on all the samples in parallel. The loaded comb with attached fragments 1010 is then dipped in the loading wells and as shown in Fig. 10D the fragments are released from the original sample templates by denaturation, using for example heat and formamide.

Fragment Generation and Loading: MicroFRA

Both the concurrent generation and loading problems are solved by the microFRA 110. The microFRA is an array of chemical micro-reactors for concurrent generation of biopolymer fragment samples for analysis. It is particularly adapted to DNA sequencing. Using any number of single tube (i.e., no separation required) DNA analysis methods, a microFRA can process DNA for analysis directly from minute, unpurified samples. This capability eliminates many manual steps, improving analysis speed and reducing errors. Integrated with the microFGE, a single instrument can perform high-capacity DNA analysis directly from raw DNA samples.

The microFRA comprises micro-reactors and capillary passages with micro-valves, which control liquid flow in the capillary passages. The construction and use of these elements is illustrated in Figs. 8 and 9 for the case of a capillary evaporative bubble valve. Fig. 8 illustrates a section of two reactors for one version of the microFRA. Typically, there would be as many reactors as sample migration lanes in the electrophoresis module. MicroFRA structural components include four silicon wafers 886a-d approximately 0.25 mm thick, reactor housing plate 888, preferably constructed from glass and approximately 3 mm thick. Reaction chambers 898 are defined in the structure along with inlet and outlet capillary passages 885, 884. Heating elements and thermocouples, not illustrated, can be incorporated in the walls of reaction chambers 898 for controlling reaction temperatures. Initial biopolymer samples and reagents are introduced into the reaction chambers through minute sample inlets 802. Additional reagents needed during fragment generation are introduced through reagent inlets 800 and capillary inlet passages 885. Biopolymer fragment samples are ejected into the electrophoresis module through outlet capillary passages 884 and fragment outlets 890.

Capillary flow in passages 884, 885 is controlled by several micro-bubble valves 894, which comprise evaporative heating elements 892 and associated electrical leads 896. Fig. 8 shows one valved inlet path and one valved outlet path connected to each reactor. Other versions can be constructed with multiple inlets and outlets to each reactor (such as may be necessary for ethanol precipitation and washing to remove salts, followed by formamide resuspension).

Each reaction chamber 898 is a truncated, conical shaped hole in the reactor housing plate 888 of depth approximately 3 mm, width approximately 1 mm, and volume approximately 1-5 µl. Construction of the microFRA involves etching in top silicon wafer 886a one semicircular capillary inlet passage 885 per reactor with diameter from 5 - 100 µm, preferably approximately 10 µm. A circular hole, with diameter approximately 1 mm, is formed in alignment with each reaction chamber. The capillary inlet passages terminate in the sides of these holes.

In second silicon wafer 886b, standard deposition techniques are used to deposit micro-heating elements 892, electrical leads 896 to the heating elements, and an electrically insulating layer protecting these components. Each capillary inlet passage is contacted by one micro-heating element. The electrical leads are brought to the edges of the wafer for making contact with external leads from the controller/power supply. The second wafer also has 1 mm holes aligned with those of first wafer and the reaction chambers. The two holes define minute sample inlet 802 into each reactor. The two wafers are bonded together and to reactor housing plate 888 as shown.

The reactor housing plate is bonded to third silicon wafer 886c, which is etched with outlet capillary passages similar to the inlet capillary passages in wafer 886a. Each outlet capillary passage terminates in vertical passage 899 which communicates with the truncated base of one reactor. Fourth wafer 886d, on which are deposited micro heating elements, associated electrical leads, and an insulating layer, similar to those of wafer 886b, is bonded to wafer 886c. Each capillary outlet passage is contacted by one micro-heating element. When the microFRA is positioned, attached, and sealed at the left of the electrophoresis module, as in Fig. 1, the outlet passages communicate with the separation medium at the heads of migration lanes. If a microFGE is used, the capillary outlet passages in wafer 886c could converge so that outlet ports 890 would match microFGE lanes 107.

Preferably, a pressure supply (not shown) is connected to reagent inlet 800 to pressure the capillary inlet passages for introducing reagents during a reaction sequence. Likewise minute sample inlet 802 into the reaction chambers can be connected to a pressure supply for forcing reaction products through the capillary outlet passages to the biopolymer fragment outlets 890.

Fragment Generation and Loading: microFRA Bubble Valve

The evaporative bubble micro-valves 894 are important to the functioning of this version of the microFRA. They provide on/off control of fluid flow in the capillary passages. Alternative micro-actuators of similar function, remote control, and ease of fabrication could be used. [See Lin et, al. Microbubble Powered Actuator, Transducers 1041 (1991).]

Figs. 9A and 9B illustrate the construction and operation of an illustrative bubble micro-valve 894. The valve comprises two wafers 985, 986 which are joined together after processing to form the structure shown in Figs. 9A and 9B. A semicircular capillary passage 988 is etched in wafer 985. A resistive heating element 992 is deposited on wafer 986, and a protective layer 993 is deposited over the heating element to prevent chemical or electrical contact with fluid in the capillary. Electrical leads, not shown, are deposited to provide for external contact. Current to the heating element is supplied through the external contacts and deposited leads from the controller/power supply. Resistive heating element 992, the electrical leads and protective layer 993 correspond to micro-heating elements 892, electrical leads 896 and the electrically insulting layer of Fig. 8 and the accompanying description.

Micro-bubble 987 obstructs the flow of fluid in this passage. It is generated by evaporating fluid in the passage with heat from resistive heating element. Cessation of the heating allows the vapor to cool and condense, collapsing the bubble, and removing the obstruction to fluid flow. Thereby an off/on valve is created and controlled by current in the evaporative heating element.

Surface tensions at the fluid/gas interfaces allow the bubble to be maintained in position despite a pressure difference across the bubble. The allowable pressure difference is determined by Laplace's equation

$$P_{fluid}=P_{vapor}+\frac{2\sigma}{R} \qquad (4)$$

where P_{fluid} is the pressure difference in the fluid in the capillary, P_{vapor} is the saturation pressure, s is surface tension, and R is the radius of the capillary passage. For a pressure difference between 1.5 and 1.0 atmospheres across the bubble, the capillary diameter must be less than approximately 10 μm.

Fragment Generation and Loading: DNA sequ ncing using microFRA and dUTP digestion

By using dUTP rich PCR primers, the microFRA can completely and automatically process DNA samples from crude DNA to labeled DNA fragments ready for separation, and eventually finished DNA sequence. Using such primers, DNA sequencing fragments can be generated simply by the sequential addition of reagents. An intermediate separation step, not easily possible in a microFRA, to remove unreacted PCR primers is not needed. The microFRA in combination with this technique eliminates all manual DNA sequencing steps. Although adapted for a microFRA, this method of making DNA sequencing fragments can be carried out in other reaction configurations.

The dUTP rich outer amplification primers are designed to prime to known vector sequences, are preferably 17-24 nucleotides long, and are synthesized with dUTP in place of dTTP. Preferably, the dUTPs are no more than 6 base pairs apart, with 4-10 dUTPs per primer molecule. The melting temperature of these primers is preferably between 54°C and 72°C.

The steps of using these primers are illustrated in Fig. 11. First, at step 1101 10,000 to 100,000 molecules of crude unpurified DNA are loaded into the microFRA reaction chambers through the minute sample inlets. Subsequent reagents can be added either through these inlets or the capillary inlet passages as convenient. No prior DNA preparation is needed. Advantageously, a sterile tip can be used to transfer colonies or other DNA sources containing single or doubly stranded DNA vector with a clonal insert directly into the reaction vessel. Second, at step 1102 amplification reagents are loaded in the reaction chambers. Amplification reagents include: 50-100 picomoles/100 μl of dUTP containing primer; 75-100 μmolar each dATP, dCTP, dGTP and dTTP; and other conventional reagents such as DNA polymerase, BSA, Ficol, and dye.

At step 1114, twenty to forty amplification cycles are performed. Each cycle comprises the steps of bringing the reaction mixture to 94°C for 5-15 secs., then to 52°C for 5 to 15 secs., and then to 72°C for 15-30 secs.

At step 1116 the dUTP rich amplification primers are removed with UDG, uracil DNA glycosylase, from the Escherichia coli ung gene. UDG removes uracil residues from both single and double stranded DNA present in the reaction mixture. Loss of the uracil residue prevents DNA base pairing and exposes the DNA sugar-phosphodiester backbone to hydrolysis into fragments containing 5′ and 3′ phosphate termini. The resulting short fragments are no longer able to hybridize to DNA and cannot form a primer for further chain elongation in the following sequencing reactions step.

Next, the reaction mixture is prepared for the sequencing reactions. The mixture is diluted 1 to 10 and a single sequencing primer, buffer and fluorescent dye labeled ddNTPs (step 1100) are added in a conventional manner. Fifteen to thirty sequencing cycles are then performed, each cycle comprising the sequential temperature steps 96°C for 5-15 secs., 50-60°C for 1 second, and extension at 60°C for 4 min. (step 1118). The DNA fragments are next ejected through the capillary outlet passages into the electrophoretic separation subsystem (step 1120). Electrophoretic separation of the DNA fragments then occurs (step 1122).

Fragment Generation and Loading: Expression Analysis

While our system has been designed to be flexible regarding biochemical design, we describe a single exemplary protocol. Recent refinements in molecular biology methods to characterize differences in gene expression makes this possible (Liang et al., 1991). The steps are as follows: (i) mRNA preparation from sample of interest; (ii) first strand cDNA synthesis; (iii) "fingerprinting" by arbitrary PCR of individual samples; and (iv) electrophoresis and fluorescent identification of differences in a single lane.

The high quality of the mRNA is assured by immediate extraction of the mRNA from fresh tissue. The mRNA is extracted from the tissue following a protocol based on the FastTrack mRNA isolation kit (Invitrogen Corp., San Diego, CA), which allows transition to purified PolyA mRNA in under 2 hours.

Complementary cDNAs are constructed by using four specific polyT primers; $d(T)_{11}VA$, $d(T)_{11}VC$, $d(T)_{11}VG$, $d(T)_{11}VT$ (V= A,C or G) to prime PolyA mRNA in four separate reverse transcriptase reactions (10ng/each). This insures that the initial PolyA mRNA pool is broken into four roughly equal portions. By constructing primers with two specific bases at the 3′ end the pool could be further divided. These methods utilize extremely small quantities of mRNA (10 ng per reaction). Reaction conditions are designed to minimize any sequence specific bias and to enhance the representation of individual species.

After sample preparation (mRNA isolation and first strand cDNA synthesis), DNA fingerprinting of the individual samples (arbitrarily primed amplification) is conducted using a cycle method based on the use of a thermostable polymerase (PCR). A series of reaction premixes, each containing a specific labeled oligonucleotide primer (one of the four polyT primers with a specific dye attached), a single arbitrary primer, nucleoside triphosphates, and Taq polymerase are added to the first strand cDNA template in an appropriate buffer. Thermal cycling follows, which generates the labeled double stranded family of products (the actual "fingerprint" consisting of 500 to 1000 fragments up to 2kb in length per reaction).

Primers are designed subject to two major constraints. The first is to insure an even distribution of priming at a specific frequency (determining the number of bands). The second is to insure specificity of the arbitrary primer (insuring reproducibility). In addition, primers are designed by searching against a human sequence database to insure that they prime at an appropriate frequency (one which will allow for the generation of the most detailed fingerprint that can be characterized within the limitations of our instrument). Arbitrary primers can be designed using mixed bases (A, T, C or G) at the 5′ end to allow larger primers to be made, while controlling both melting temperature (all combinations have same melting temperature) and specificity, and with a fixed 3′ end (conveniently having a restriction enzyme site to speed up later cloning).

To facilitate the direct identification of the nature of the coding region of the differentially expressed genes, an arbitrary primer strategy which does not utilize the common 3′ PolyT primer is used. In this case two arbitrary primers (one of which is labeled) are used for the amplification step.

Analysis Computer and Signal Analysis

Analysis computer 112 is a conventional computer including a programmable processor and both short and long term memory. For example, an Intel 80486 or higher DOS/Windows compatible computer is adequate. An Intel 80486 33 mHz with 16 MB of RAM and 500 MB hard drive is exemplary for both control and analysis. Its control functions required during an analysis run have been previously described. Additionally, it performs the signal analysis which determines biopolymer sample characteristics from a record of the separated fragment samples. The analysis method and apparatus comprises several steps sequentially executed by the processor, each step using input stored in memory and producing output also stored in memory. The data storage memory can utilize either magnetic or electronic memory as appropriate for storing intermediate results between steps. If the microFGE′s sample shunting capability is used, data analysis must be done during an analysis run to identify particular samples of interest to shunt. Otherwise, analysis can be done at any time.

The version of the analysis method described and illustrated is directed to determining a DNA base sequence from electrophoretic separation of Sanger sequencing reaction fragments. In this application, four fluorescent dye labels chosen to have distinguishable emission peaks must be recognized. However, the techniques can be applied to analyses of other types of biopolymers. In particular, fragments from more than one sequencing reaction can be loaded into a single migration lane, and distinguished through the use of more than four fluorescent dye labels, such as five or eight or more. The signals associated with each of the reactions can be distinguished and processed independently whereby, for example, a single migration lane can be made to serve the function of two.

The analysis method must be adapted to the microFGE electrophoretic module and its running conditions. Because small migration lanes carry small fragment samples, the microFGE generates lower intensity signals with a lower signal to noise ratio than conventional electrophoretic modules. Also, the microFGE's short lanes and high voltages result in more rapid presentation of fragment samples and less clearly defined fluorescence peaks. Further, detailed variations in running conditions due to gel characteristics, voltage used, sample analyzed, and so forth, require that the method be trainable to these variations. These and other characteristics of the microFGE require the uniquely adapted analysis described below in order to achieve better than 99% recognition accuracy.

Fig. 12 is a high level flow chart of the analysis. Raw signals from each detector element at each observation time are gathered by transmission imaging spectrograph 100 (Fig. 1) and stored in memory at step 1229. The signal intensity from adjacent detector elements of CCD array 228 may be grouped or summed, called "binning", into sets, called "bins", and the cumulated value of the set reported. Binning done on the CCD array is controlled by software supplied with this component and is dynamically adjustable. Further binning is done by preprocess step 1230. Preferably, 256 spatial bins each spanning four detector elements are defined (the detector having 1024 total elements along the spatial axis) Each migration lane is assigned to one spatial bin at each observation time, the spatial bin associated to each migration lane being substantially the same for all observation times throughout a run, the spectrograph thus allowing simultaneous detection of up to 256 lanes. In the preferred embodiment of the invention, each spatial bin is subdivided into four spectral intensity bins, each spanning 40 detector elements (the detector having 256 total elements along the spectral axis) centered on the emission maximum of the four dyes used to label the four ddNTP bases. Obviously, additional spectral bins can be accommodated by the 256 CCD elements along the spectral axis; and by reducing the number of elements per spectral bin and/or using larger arrays to increase the total number of elements along the spectral axis, the number of spectral bins can readily be increased to about 16 or so, permitting the simultaneous detection of as many different fluorescence signals from different dye labels. The binned signals are further preprocessed at step 1230 by removing recognizable noise and outputting separately into memory the spectral intensity data for each migration lane for each observation time.

Basecalling step 1232 compares the spectral intensity data for each migration lane for each observation time against an event prototype file.

Event prototype file is generated by training processing at step 1234. For example, a DNA sample whose sequence is known with very high confidence is analyzed in the electrophoretic module by collecting the fluorescence from each of the four fragment labels and generating spectral intensity data that is stored in memory. In particular, the preprocessed spectral intensity sequences from the migration lane with the known sample are tagged at step 1235 with the known base events - A, C, T, G, or the null event X - at the observation times at which the known bases generate signals. This may be done manually or automatically. Then, for all events of each of the different base types, the local time behavior of the signal is averaged, or clustered at step 1236 to generate a prototype intensity signal trace for each event. The prototypes are stored in memory at step 1237 as the event prototype file.

In the preferred embodiment of the invention, event prototypes are determined for pairs of recognition events. Since there are four base events and the null event, there are 16 (=4x4) different pairs of non-null events and therefore 16 different prototype intensity signal traces. Other choices of events are possible with this method.

The basecalling step compares the time series of the preprocessed signals from the spectral intensity bins in a spatial lane 107 with prototype intensity series. If the observed series is judged by some measure to be close to a prototype series, the basecalling step recognizes the base known to be associated with that prototype series. The recognized base identities are output to memory at step 1242 as the nucleotide sequence for that lane 107. This sequence can be finally output at step 1243 or further postprocessed at step 1244. In addition, the measure by which the observed series is judged to be close to the prototype series serves as an indication of the confidence which should be assigned to the accuracy of the basecall. This can be output for every base, along with the base sequence, and can further be output for every possible prototype at every basecall. In the case that multiple samples are analyzed using a single migration lane, it is likely that the samples will be related, for example, the forward and reverse sequences of a 2kb clone can be analyzed simultaneously. In this case, information linking the two samples (in the example: the fact that they are at the extremes of a 2kb clone) is also output.

In postprocessing, if partial sequence information for the DNA sample is known a priori, for example sequences of vector DNA, step 1238 recognizes and trims them from the output sequence. Subsequently a Monte Carlo proofreading step 1240 is executed. Proofreading involves checking the global consistency between the basecalling output and the original unprocessed data. Special knowledge about the DNA being analyzed, for example that the DNA codes for a protein, can also be supplied as at step 1241.

In genomic scale sequencing, it is desired to assemble and align the base sequence of many fragments into the base sequence of an entire genome. The closeness to prototypes output at step 1243 can be used in dynamic programming (minimum edit distance) alignment programs which are known in the art, in order to increase the accuracy of the alignment. Bonfield, J.K., et al., "The Application of Numerical Estimates of Base Calling Accuracy to DNA Sequencing Projects, Nucleic Acids Res., 23, 2406-1410 (1995), which is incorporated herein by reference. Additionally, information relating multiple samples analyzed in a single migration lane can also be used to enhance the assembly of large contigs of known sequence, by providing links (in the example above--forward and reverse strands separated by 2 kb) which can be used to join isolated contigs. Even if the analysis of the second sample is not complete (for example only a single base termination reaction is carried out in the reverse direction, and only a fifth fluorescent dye is used), the information output at step 1243 can still be used to advantage in large-scale contig assembly.

Data Analysis Method: Preprocessing Step.

Fig. 13 is a detailed flow chart of preprocessing step 1230. The input from transmission imaging spectrograph 100 is a concatenation of signals from consecutive exposures of the CCD camera. Each exposure produces binary data representing charge intensities at individual pixels (and accumulated intensities in on-chip defined bins). The pixels are grouped into spatial and spectral bins as previously described, each spatial bin having four associated spectral bins. All further processing is done on these binned signals. First the spatial bin assigned to each migration lane is identified (step 1440) and a file is created in memory for each lane (step 1442). For each migration lane, the operator chooses one of the 256 spatial bins to best represent the fluorescence emitted by samples in that lane. All remaining processing then continues independently for each lane.

Next, for each lane, recognizable noise is removed by high and low pass filtering. Spikes, which are one observation time anomalies, are removed (step 1444) by replacing a signal value in any spectral bin at any observation time with an average of the signal values in the same spectral bin at the preceding and succeeding observation times if the value differs drastically from that average. Next, the background signal is identified and subtracted (step 1446). For each observation time and spectral bin, a background value is computed and subtracted. The background is the best linear fit to the absolute signal minima taken from four windows near the observation time in that spectral bin. The first window contains enough future time points to include preferably about 10 base recognition events (or peaks); the second window enough for 20 future events; the other windows include 10 and 20 past events. The filtered signals are stored in memory at step 1447.

Next, for each observation time, a linear conversion is made from fluorescence intensity signals to signals representative of dye concentration (step 1450). This is done by multiplying the 4 spectral bin values in the data stored at step 1447 by a 4X4 conversion matrix to obtain 4 new values representative of the four dye concentrations. This matrix is determined at step 1448 prior to the conversion in the following adaptive manner. The signals stored at step 1447 are scanned. For a range of observation times from the middle of the analysis run, preferably the middle 1/2, during which range each signal peak is influenced by a single base event, the three highest peak values are found in each spectral bin. This is done by finding a first maximum, excluding a window around that maximum, then similarly finding a second and third maximum. These peaks are taken to correspond to existence of a single dye in the detection region. (Validity of the assumption is tested by comparing the shapes of the dye emission curves with the ratios of signal intensities in the spectral bins.) For each of the three highest peaks of each of the four dyes, the values in the four spectral bins are obtained. For each bin and each dye the three values are averaged to obtain a set of four numbers that represents the ideal fluorescence signature of that dye. The four signatures are assembled as the rows of a 4X4 matrix. For example, an illustrative signature matrix might be:

	measured average fluorescence intensity level in bins associated with:			
Nucleotide	A	T	G	C
A	800	100	50	100
T	300	700	100	50
G	100	200	900	200
C	50	50	300	800

The inverse of this matrix is the desired linear conversion factor input to step 1450.

Alternatively, a matrix can be recalled from memory which has already been generated on the basis of knowledge of the emission spectra of the fluorescent dyes. Instrument spectral alignment variability can be accomodated by choosing the best of a collection of several such matrices stored in memory. The choice of the best matrix is made by testing the extent to which negative values are generated as the output dye intensities, with the optimum matrix having the minimum accumulation of negative dye intensities.

Alternatively, more than four dyes can be employed and a corresponding number of binning regions can be used to accumulate the fluorescence signals from such dyes. In the case that the number of binning regions exceeds the number of dyes, a best-fit linear conversion or pseudo-inverse can be found to determine the dye concentrations.

In the case of multiple samples, such as the forward and reverse reactions, being analyzed in a single migration lane, an analogous 8x8 matrix analysis replaces the 4x4 analysis described above.

An illustrative signature matrix in this case might be:

Nucleotide	C 1	A 1	G 1	T 1	C 2	A 2	G 2	T 2
C sample1	930	430	0	0	0	0	0	0
A sample1	790	1080	90	0	0	0		0
G sample1	500	680	1090	380	0	0	0	0
T sample1	320	420	760	1090	380	0	0	0
C sample2	230	260	530	810	950	70	0	0
A sample2	0	0	260	470	740	1070	420	0
G sample2	0	0	140	270	500	780	1090	330
T sample2	0	0	70	130	230	430	690	1060

Finally at step 1452, the signal values at consecutive observation times are added into one new observation and output to memory. Consecutive observations, or larger adjacent groups of observations, are additively combined so that approximately five resultant observation times occur between consecutive base recognition events.

Alternatively, this renormalization of the time sampling interval can include interpolation, and can be carried out independently for each of the different fluorescent dyes in order to accomodate the different influence that the dyes themselves have on the mobility of the DNA fragments.

Data Analysis Method: Basecalling Step

Basecalling (step 1232) recognizes the event of a labeled fragment in a migration lane passing through the laser beam 113 and discriminates the event into one of a set of classes according to the dye label carried by the fragment. Four initial choices must be made: a configuration space to represent recognition events; a mapping of signal traces into paths in this configuration space; the location of events in the configuration space; and a criterion for determining when the configuration space path represents an event. First, this method is schematically illustrated for a simple case, then the preferred version is described.

Figs. 14A and 14B schematically illustrate a simple, exemplary case. In Fig. 14A are two signal traces: one 1328 having a single, tall peak 1326; and a second 1332 having a single, broad, low peak 1330. Trace 1328 represents an event; the trace 1332 represents only noise.

Fig. 14B illustrates the mapping of the signal values at the preceding time point, t-1, at the current time point, t, and at the succeeding time point, t+1, into single 3-dimensional points (t-1, t, t+1) in a 3-dimensional configuration with these three values for coordinates. Thus each point in configuration space represents a triplet of three consecutive signal intensities. Next, based on knowledge of prior event characteristics, an event prototype characteristic of detection of passage of a particular dye label fragment through the laser beam is assumed to be located at 1322. For event recognition, the configuration space path must pass within sphere 1340 about the event prototype.

Signal trace 1328 maps to loop 1324 in configuration space, beginning and ending at the origin and passing, in the example shown, within the recognition sphere. It therefore represents an event. Signal trace 1332 illustratively maps to loop 1342 in configuration space, which does not pass within the sphere. It is therefore not recognized as an event. In this manner, events are recognized and discriminated.

Bas calling Step: Embodiment

As indicated in the discussion of Fig. 12, event prototypes are determined for pairs of recognition events, there being sixteen such pairs corresponding to the sixteen doublets of DNA bases -- CA, CC, CG, CT . . . TA, TC, TT, TG.

The event recognition criterion is that a local minimum occurs in the distance between the signal trace as mapped into configuration space and one prototype event. Starting from the previous base recognition event, and stepping forward observation by observation, the configuration space distance to each of the 16 prototype events is computed at each observation time. The event identity with the smallest distance and that distance value are saved. If the closest prototypes at the current and adjacent observations are the same, and if the current distance to that closest prototype is less than the distances at adjacent observations, then that prototype is recognized. As indicated above, there are approximately five observation times between successive base recognition events. Any time a prototype is recongnized, the distance to each of the 16 prototype events can be stored for future analysis or output.

In essence the basecalling step measures at a series of observation times following a base recognition event the correlation between the dye concentration values derived from four signals received at the four spectral bins and the corresponding dye concentration values associated with the sixteen doublets of DNA bases that have previously been stored in the prototype file. Beginning with the first observation time following a base recognition event, the measurement is made by calculating a weighted sum of the squares of the differences between five successive time samples of the dye concentration values derived from the four received signals and five successive time samples of the corresponding signals of each of the sixteen doublets, repeating the calculation for the next set of five successive time samples of dye concentration values displaced by one observation time from the previous calculation and the same set of five samples of each of the sixteen doublets, and so on. The distance at the central sample point is weighted highest (2.0); the distances at the previous and succeeding points are weighted intermediately (1.5); and the remaining distances are not weighted (1.0).

The general form of the equation for the weighted sum of the squares is

$$\begin{array}{rcr}
(TD_{-2} - TP_{-2})^2 & + & 1.5\cdot(TD_{-1} - TP_{-1})^2 \\
+2\cdot(TD_0 - TP_0)^2 & + & 1.5\cdot(TD_{-1} - TP_{-1})^2 \\
+(TD_{+2} - TP_2)^2 & + & (AD_{-2} - AP_{-2})^2 \\
1.5\cdot(AD_{-1} - AP_{-1})^2 & + & 2\cdot(AD_0 - AP_0)^2 \\
1.5\cdot(AD_{+1} - AP_{+1})^2 & + & (AD_{+2} - AP_{+2})^2 \\
(GD_{-2} - GP_{-2})^2 & + & 1.5\cdot(GD_{-1} - GP_{-1})^2 \\
2\cdot(GD_0 - GP_0)^2 & + & 1.5\cdot(GD_{+1} - GP_{+1})^2 \\
(GD_{+2} - GP_{+2})^2 & + & (CD_{-2} - CP_{-2})^2 \\
1.5\cdot(CD_{-1} - CP_{-1})^2 & + & 2\cdot(CD_0 - CP_0)^2 \\
1.5\cdot(CD_{+1} - CP_{+1})^2 & + & (CD_{+2} - CP_{+2})^2
\end{array} \qquad (5)$$

where the first term in each squared expression is the sample of the received signal and the second term is the sample of the stored prototype signal, the letters T, A, G, C identify the relevant dye concentration and the subscript indicates the sample number and its order in time.

The value of the above equation is calculated for each of the sixteen doublets for each of the observation times until a closest prototype is located. Alternatively, however, it is not necessary to make the calculation for twelve of the sixteen doublets because the identity of the first nucleotide in the doublet is already known from the immediately previous basecalling step.

In addition, each calculated value of the sum of the squares is weighted by a factor that increases with the time between the actual observation time and the expected time of the next base recognition event. A match is identified at the observation time where the weighted sum of squares is determined to be lowest.

Further details of the base calling step are as follows:

The configuration space is a composite of a 20-dimensional signal-intensity subspace and a 1-dimensional time-from-event-recognition subspace. Signal traces map into the signal-intensity subspace by assigning for the 20 coordinates, sequentially, the four spectral bin values at each of the five observation times - the twice previous time, t-2, the previous time, t-1, the current time, t, the succeeding time, t+1, and the twice succeeding time, t+2. This maps adjacent portions of the signal trace to a 20-dimensional vector in this subspace at the observation time, t. In the 1-dimensional time-from-event-recognition subspace, the coordinate is assigned to the time difference between the current time and the time at the last recognition event.

The distance in the configuration space is the product of distances computed separately in the two subspaces. In the 20-dimensional signal-intensity subspace, the distance is a weighted sum of the squares of the distances (sum of squares of signal coordinate differences) between the signal and a prototype point at the five time points. The distance in the 1-dimensional time-from-event-recognition subspace is the sum of 1.0 and the weighted (0.3) square of the difference between the coordinate value in that subspace and the average time between basecalls.

This precise calculation is illustrated by the following C++ code:

```
class datapoint {
  double c, t, a, g;  /* normalized fluorescence values */
  int tag;            /* call for this data point */
};

class vector {
  datapoint twoprev;     /* data point at current time - 2 */
  datapoint prev;        /* data point at previous time */
  datapoint curr;        /* data point at current time */
  datapoint next;        /* data point at next time */
  datapoint twonext;     /* data point at current time + 2 */
```

```
   int lastcall;          /* base last called */
   double timetocall;     /* time since last base call */
   int tag;               /* call for this vector */
  };

 dist = (
     (
     pow((vec->twoprev.c - average.twoprev.c), 2) +
     pow((vec->twoprev.a - average.twoprev.a), 2) +
     pow((vec->twoprev.g - average.twoprev.g), 2) +
     pow((vec->twoprev.t - average.twoprev.t), 2) +
     pow((vec->prev.c - average.prev.c)*1.5, 2) +
     pow((vec->prev.a - average.prev.a)*1.5, 2) +
     pow((vec->prev.g - average.prev.g)*1.5, 2) +
     pow((vec->prev.t - average.prev.t)*1.5, 2) +
     pow((vec->curr.c - average.curr.c)*2.0, 2) +
     pow((vec->curr.a - average.curr.a)*2.0, 2) +
     pow((vec->curr.g - average.curr.g)*2.0, 2) +
     pow((vec->curr.t - average.curr.t)*2.0, 2) +
     pow((vec->next.c - average.next.c)*1.5, 2) +
     pow((vec->next.a - average.next.a)*1.5, 2) +
     pow((vec->next.g - average.next.g)*1.5, 2) +
     pow((vec->next.t - average.next.t)*1.5, 2) +
     pow((vec->twonext.c - average.twonext.c), 2) +
     pow((vec->twonext.a - average.twonext.a), 2) +
     pow((vec->twonext.g - average.twonext.g), 2) +
     pow((vec->twonext.t - average.twonext.t), 2)     ) *
     (pow(0.3*(vec->timetocall - average.timetocall), 2) + 1)
       );
```

Prototype events for each of the sixteen doublets of DNA bases -- CA,CC,CG,CT,...,TA,TC,TT,TG are stored in the prototype file at step 1237. This file is mapped into the 20-dimensional signal-intensity subspace and 1-dimensional time-from-event-recognition subspace. Then all (20+1)-dimensional vectors at which a base is recognized are assembled according to which doublet is formed by the current and previous basecall. Vectors for each doublet are averaged arithmetically to form a prototype. The vector averages are output to memory.

A flow chart for the basecalling step is shown in Fig. 15. The basic processing loop is entered at step 1556; the next observation is input from memory; and a new vector is mapped in configuration space. Variables are initialized at step 1576, by looking forward a sufficient number of observations into the input data. Distances in the 20-dimensional signal-intensity subspace to all prototype events are computed at step 1558. The distance in the 1-dimensional time-from-event-recognition subspace is computed at step 1560. The two distances are multiplied at step 1562 to give the configuration space distance. The local distance minimum event recognition criterion is evaluated at step 1566. Illustratively, a local minimum is recognized when the path in configuration space has been nearest to a single prototype for at least three time points and the distance to that prototype at one time point is less than the distances at adjacent time points. If no event is recognized, the method returns to step 1556. If the criterion is met, that doublet event is recognized and saved at step 1570. Since the prior base recognized has been saved at step 1570, it and the currently recognized doublet are used to determine the current base at step 1572. This base and its recognition time is output at step 1242. Next (optionally), the average time between recognition events is updated by computing a moving average of the time between events. Adjustments from this average are made for known differences in electrophoretic mobility dependent on DNA sequence. Since the average time between basecalls depends on the nature of the separation gels, the voltage used, and other running conditions, it can be expected to vary from run to run. The average time between basecalls can also vary within a given run from start to finish.

Monte Carlo Proofreading St p

The optional postprocessing consists of trimming known sequences at step 1238 then Monte Carlo proofreading at step 1240. Trimming known sequences includes removing known sub-sequences, usually vector DNA, from the processed data input from 1242.

Proofreading seeks to improve the overall match between the signal intensities and the recognized base events. The basecalling step looks locally at groups of observations representing two base recognition events seeking local minima. Proofreading tests the recognition globally by making proposed alterations (moves) and testing whether recognition accuracy is ultimately improved by the alterations. In this process, known restrictions on the DNA, such as it being a protein code, can be utilized. This is an important step for improving recognition accuracy. However, since it requires data from an entire analysis run, it cannot by used for sample selection and shunting.

Fig. 16 provides flow charts of the Monte Carlo proofreading step. For conventional Monte Carlo techniques, refer to Press, et al., Numerical Recipes in C (1988), which is herein incorporated by reference. Monte Carlo proofreading requires three initial choices: a set of sequence alterations to try, an energy function to evaluate success of the alterations, and an annealing schedule to exercise overall control on the proofreading. The following are preferred choices. Choose for the set of alterations at an observation time: insert a new base recognition, delete a base recognition, move the nearest base recognition forward one observation time, or move the nearest base recognition backward one observation time. Other sets of alterations may incorporate specific knowledge about the DNA sample. For example, alterations should be limited to valid protein codons if the DNA is known to code for a protein. Choose for the energy function the sum over all base recognition events of the square of the distance in configuration space between the prototypes of the recognized base sequence and corresponding observation vectors. For the annealing schedule, choose a simulated temperature decay exponential in the number of epochs of the proofreading method, an epoch being a certain number of iterations of the alter-and-test loop. The simulated temperature probabilistically controls acceptance.

Proofreading begins at step 1610 with the choice of a temperature comparable to the value of the initial energy function. (Units are chosen so that the Boltzman constant is 1.0.) Next an epoch of proofreading is run at step 1612. The temperature is exponentially decremented at step 1614, by multiplication with a decay constant less than 1.0, and compared to a minimum. The decay constant determines the number of proofreading epochs to execute. If the temperature exceeds the minimum as tested at step 1616, the method loops back to step 1612. If not, the method ends at step 1616 and the base sequence with all permanently incorporated alterations is output to memory. The analysis method is complete.

The procedural steps in execution of one epoch of proofreading follow. The input data in memory is the base sequence output from the basecalling step 1242, as trimmed at step 1238, and the preprocessed signal traces 1228. The alter-and-test loop begins with selection of a random observation time from the sequencing run at step 1686 and a random sequence alteration from the chosen set of alterations at step 1688. A new energy is computed at step 1690 from the base sequence using the temporarily incorporated alteration and the input preprocessed signal traces. The new energy is tested at step 1692. If the new energy is lower than the previous energy, the alteration is permanently incorporated in the base sequence at step 1600. A convergence stop condition is tested at step 1602, which is preferably a certain number of alter-and-test iterations. Other stop conditions are possible, such as a certain energy decrement during the epoch. If the new energy is not lower, th move is allowed or disallowed probabilistically according to the

Boltzmann criterion. A random number is generated at step 1694 to test the Boltzman probability of this move at step 1696. The Boltzman probability is determined by:

$$\textit{Acceptance Probability} = \exp(-\frac{\textit{energy change of move}}{T}) \quad (6)$$

where T is the current temperature of the epoch set at step 1614. If the move is allowed as tested at step 1698, it is permanently incorporated in the base sequence by step 1600. In either case the stop condition is again tested. If the stop condition is met, the epoch ends and overall stop condition at step 1616 is tested.

The following examples are illustrative of the application of the present invention.

Example 1 - Imaging Spectrograph and Analysis Method

A segment of double stranded DNA supplied as control with reagents from Applied Biosystems Inc. (Foster City, CA) (pGEM -3Zf(+) from the -21M13 forward primer) was analyzed.

Ultrafloat glass (with a green tinge) was used as the bottom plate. BK7 glass was used as the top plate. A 100 micron polyester spacer gasket separated the two pieces of glass. Bind silane, consisting of 1 milliliter of ethanol (J.T. Baker; Phillipsburg, NJ), 5 microliters of gamma-methacryloxypropyltrimethoxysilane (Sigma Chemical Company; St. Louis, MO), and 50 microliters of 10 percent acetic acid (EM Science; Gibbstown, NJ), was applied sparingly to each edge of glass which contacted the comb. The polycarbonate comb used had physical dimensions of 0.75 millimeter thickness, and teeth making wells in the gel spaced on 2.25 millimeter centers. Gel was 5 percent monomer 19:1 acrylamide:bisacrylamide Sequagel (National Diagnostics; Atlanta, GA) with 8.3M urea. The running buffer was 1x Tris-Borate-EDTA. The gel was allowed to polymerize for 3.5 hours, and was prerun for 0.5 hour. The sample was resuspended in 3-6 microliters of formamide/EDTA load solution. 0.5 microliters of sample were loaded into the gel.

The collection lens was a 250 mm f5.6 Zeiss medium format telephoto lens. Further description of the spectrograph is provided in the detailed description of the invention. Laser power of 82 milliWatts from an LS1000 argon ion laser (American Laser Corporation; Salt Lake City, UT) was filtered to select the 515 nanometer wavelength using a laser line filter, resulting in about 35 milliWatts focused through the side of the gel. The electrophoresis path from the loading region to the detection region had a length of 23 or 24 centimeters. Exposure times were 2 seconds per frame. Detector read time was roughly 0.1 seconds. 4000 total frames were collected. Electrophoresis was conducted at 2500 Volts constant voltage applied across 28.5 centimeters using an EC 650 (E-C Apparatus Corporation; St. Petersburg, FL) power supply. This resulted in dissipation of 12.3 Watts in the gel. The circulated water was kept at 40 C. Samples were injected for 15-30 seconds.

The transmission imaging spectrograph recorded the fluorescence emitted by the labeled fragments. The seven lines of Fig. 17 show the trace of the preprocessed fluorescence intensities of the four dye labels as a function of time for one migration lane as output from the preprocessor. The letters underneath the time axis are the determined basecalls. Comparing to published data, from the center of the first line on there was perfect agreement, with the exception in the last line of one missing T and G (from a GGGG sequence). Correct functioning of the spectrograph and the analysis methods was demonstrated.

Example 2 - MicroFGE

Analyses were run with two conventional glass plate modules and the microFGE. The first conventional module has ultra-thin gel, with a 80 μm plate separation and a 23 cm migration path; the second is ultra-thin with a 80 μm separation and a 10 cm migration path. Both have 32 loading lanes formed with 2.25 mm center spacing. The microFGE has 80 channels, 80 micron deep on 1.125 mm center spacing and a 10 cm path.

The electrophoresis modules were loaded with 5% or 6% (19:1) polyacrylamide gels with 8.3 M urea. Well-forming combs formed the loading wells in the loading region 463. Bind silane (as above) assisted adhesion of gel to glass in the loading region. The gel was allowed to polymerize for 3 hours, at which point the well comb was removed leaving loading wells placed at the head of each migration lane. The running buffer was 1 X TBE. The gel was then heated to a heat exchange temperature of 40° C.

Biopolymer fragment samples of a segment of M13 DNA were prepared, separated from the sequencing reaction medium, and resuspended in 3 µl of loading solution. For the microFGE 50-100 nl of the loading solution and for a conventional glass plate 400-500 nl were loaded into the loading wells. With a conventional glass plate, the 23 cm path from the loading region to the detection region resulted in separation of about 400 DNA bases in 2.5 hours with 2500 volts applied over 28.5 cm.

Figs. 18A, 18B and 18C show the fluorescence traces from the transmission imaging spectrograph. In Fig. 18A the trace is from the glass plate module with ultra-thin plate spacing (90V/cm) and 23 cm path length. In Fig. 18B the trace is from the glass plate module with ultra-thin spacing at a 10cm path length (100 V/cm). In Fig. 18C the trace shows results obtained with the 10 cm path of the microFGE (100V/cm).

Figs. 19A and 19B illustrate the output of the CCD array. They demonstrate the ability of the array to discriminate the signals from the different migration lanes and from different dyes.

Integrated functioning of the microFGE with the spectrograph were demonstrated. Further, the traces of Figs. 18A, 18B, 18C, 19A and 19B provide evidence that: 1) a single well at the head of a microFGE lane can be loaded with a DNA fragment sample; 2) the DNA fragments are separated electrophoretically as they travel down a single lane in the microFGE; 3) the fragments exit the lane and move into the laser channel where they can be excited to fluorescence and imaged by the transmission imaging spectrograph; 4) the spatial broadening associated with exit from the lane is less than the spacing between the microFGE lanes; and 5) the dyes associated with the different nucleotides can be distinguished in time sufficient for base resolution.

As will be apparent to those skilled in the art, numerous variations may be made in the practice of our invention.

This invention is not limited to the use of any single sequencing chemistry; both chemical and enzymatic methods are enabled. By way of example, enzymatic methods can be used that do not rely on the use of chain terminator chemistry. Sequencing in an integrated device may be enabled by sequence ladder generation techniques other than Sanger methods. A coupled procedure can be used that will rely on the generation of PCR-amplified products and subsequent direct generation of ladders by exonuclease digestion. This is made possible by the incorporation of blocking base analogs for A, C, G, and T that allow for the PCR extension to proceed to completion in each cycle, but that cause 5′ to 3′ activity of single strand exonucleases (e.g. Exo I; New England Biolabs; Beverly, MA) to be blocked. After four separate PCR reactions, each with a single modified blocking base, the products can be digested with an exonuclease, thereby producing a set of nested fragments terminating at each point of incorporation of the blocking base analog. Suitable use of dye labeled primers allows for the identification of the blocking base which terminates the fragment. These blocking bases are made by incorporating substitutions into the chemical structure of the DNA bases which allow the base to be incorporated by enzymatic action in a growing DNA strand, and also allow such strand to be a template for growth of a complementary strand (DOE Human Genome Project Report, Spring 1995). Use of non-chain-terminating fluorescent labels attached to the blocking base analogs would enable reactions performed in a single vessel. Biotin immobilization can be used on one primer to allow strand separation and separate analysis of the fragment ladders generated from each complementary strand.

Chemistry techniques producing sequence fragments from both complementary strands of DNA also are enabled by our invention. Chemistry techniques for generating ladders of sequence fragments have previously been used to generate fragments from,and therefore information about a single strand at a time. With the full spectral capability of the instrument, and with the analysis capability of the software, it is possible to resolve sequence information directly generated from both strands simultaneously, thereby increasing the accuracy, robustness and reliability of analysis of a biopolymer sample. In one example, two different primers are used for the two ends of the two strands of DNA. Standard Sanger fragment ladders are generated corresponding to both strands along the DNA in a single reaction, with opposite directions (strands) having unique dye labeled primers. The dyes are chosen to be spectrally resolvable. A strategy incorporating four dyes readily allows both strands of a fragment to be sequenced in two lanes, each lane corresponding to sequence information on two bases (e.g. C and T) for each of the two strands. Advantageously, a strategy incorporating eight spectrally resolvable dyes allows simultaneous independent analysis of both sequencing ladders in a single lane. Binary coding strategy can be used to decrease the number of dyes required to perform the same simultaneous independent analysis of a biopolymer fragment.

Even if the analysis of the second sample is not complete (for example only a single base termination reaction is carried out in the reverse direction, and only a fifth fluorescent dye is used), the information can still be used to advantage in large-scale contig assembly.

A variety of solid-phase supports can be used to bring either reactants, template or product into or out of the sample loading wells or microfabricated reaction vessels. Products and templates can be coupled to the support either covalently or non-covalently. Examples of non-covalent attachment are (streptavidin-biotin), and (antibody-small epitope). Hybridization between complementary strands of DNA is an alternative non-covalent attachment means. Covalent attachments made via disulfide bonds are also useful; release of the attached species is accomplished by a change in reduction potential resulting in the break of the disulfide bond. Techniques eliminating the need for chemical separation steps in the reaction process are ideally matched to the invention, and are enabled, for example, by solid-phase magnetic separations. Specifically, minute magnetic beads (Dynal Corporation) are used in standard biochemical protocols for material transport, and are a suitable substitute for the fixed teeth of a comb used for solid-phase loading. Streptavidin coated magnetic beads can be processed in the same manner as the combs. Minute magnetic bead allow for quantitation of sample transport and are suitable for loading means based on mechanisms for moving magnetic particles (C.H. Ahn and M.G. Allen, "A Fully Integrated Micromachined Magnetic Particle Manipulator and Separator", J. of Micromechanical Systems, 2, 15-22 (1993)). The transport of reaction products on magnetic beads also allows for the concurrent separation of reaction products and unreacted reagent mixtures.

Numerous variations may also be practiced in the signal processing used to identify the nucleotides and the same techniques may be used in other signal matching applications. For example, the comparisons may be made using data representative of triples of nucleotides instead of pairs of nucleotides; and other matching strategies may be used.

APPENDIX

The following computer program listings are copyright 1995 CuraGen, Inc. © 1995 CuraGen, Inc.

```
/* Vector and cluster classes */
/* In numeric tags, 0=C, 1=A, 2=G, 3=T, -1=X */

extern "C" {
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
}

#define NUMCLUSTS 16

double currmin;
int currcluster;

class cluster;

class datapoint {
 public:
  double c, t, a, g;  /* fluorescence values */
  int tag;            /* call for this data point */
  void print(FILE* = stdout, int = 0);
  int input(FILE* = stdin, int = 0);
  void init(double, double, double, double, int = -1);
  double fluorescence(int); /* returns value of argument channel */
};

class vector {
 public:
  datapoint threeprev; /* data point at current time - 3 */
  datapoint twoprev; /* data point at current time - 2 */
  datapoint prev;      /* data point at previous time */
  datapoint curr;      /* data point at current time */
  datapoint next;      /* data point at next time */
  datapoint twonext; /* data point at current time + 2 */
  datapoint threenext; /* data point at current time + 3 */
  int lastcall;        /* base last called */
  double timetocall;   /* time since last base call (double for averaging) */
  double lastcallval;  /* value of base channel at previous call */
  double max;          /* max fluorescence value */
  double twoprevmax;   /* max for twoprev */
  double twonextmax;   /* max for twonext */
  int tag;             /* call for this vector */
  void normalize();    /* normalizes fluorescence values */
  int findclust();     /* calculat s index of cluster for clustering */
  int call(cluster **); /* determines call of closest cluster */
  void print(FILE* = stdout);
  void input(FILE* = stdin);
};

class cluster {
 public:
  vector data[64];        /* vectors in cluster -- should be dynamic */
  int size;               /* number of vectors in cluster */
  vector average;         /* average for cluster */
  void addvec(vector *);  /* add vector to cluster */
  double distance(vector *);  /* determine distance from vector to average */
  void create_average();      /* create average from data */
};

int iscall(int);
int calltoint(char);
char inttocall(int);
double avedist(int,int, cluster **);
int clusterprevtag(int);
int clustercurrtag(int);
void averagetwovecs(vector *, vector*, vector *);
void averagetwodps(datapoint *, datapoint *, datapoint *);
```

```
extern "C" {
   #include <math.h>
}

void invert(double **, double **, int);
```

```
#include "cluster.h"

#define TAGGEDFILE 1
   /* if 1, expects input file to be tagged with character tags */

/* Call data by closest cluster. */

extern double currmin;
extern int currcluster;

int main(int argc, char *argv[]) {
  int time=0;        /* current time */
  int basenum=0;     /* number of bases called */
  int timeofcall;    /* time of last base call */
  int lastcall;      /* last base call */
  double lastcallval; /* value of call channel at last call */
  int i;
  int callmistakes = 0;

  FILE *datafile;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <data file>\n", argv[0]);
    exit(1);
  }

  if (!(datafile = fopen(argv[1], "r"))) {
    fprintf(stderr, "Can not open data file %s.\n", argv[1]);
    exit(1);
  }

  cluster *clust[NUMCLUSTS];  /* clusters */
  cluster *rv;

  /* read through file*/
  for (i = 0; i < NUMCLUSTS; i++) {  /* initialize */
    if ((rv = (cluster *) malloc(sizeof(cluster))) != NULL) {
      clust[i] = rv;
    }
    else {
      fprintf(stderr, "\nNot enough memory - Cluster %d\n", i);
      exit(1);
    }
  }

  datapoint threeprev, twoprev, twonext, threenext;
datapoint prev, curr, next; /* previous, current, and next data points */
vector vec;                 /* current vector */


char call;
int numclusts = 0, xclusts = 0, cclusts = 0, tclusts = 0, aclusts = 0;
int gclusts = 0;
int shiftflag = 0;

/* initialize threeprev through twonext to first six time points */
threeprev.input(datafile, TAGGEDFILE);
twoprev.input(datafile, TAGGEDFILE);
prev.input(datafile, TAGGEDFILE);
curr.input(datafile, TAGGEDFILE);
next.input(datafile, TAGGEDFILE);
twonext.input(datafile, TAGGEDFILE);
time = 2;
lastcall = 0;
lastcallval = 1;
timeofcall = 0;

/* read through */
fprintf(stderr, "Calling data.\n\n");

while ((threenext.input(datafile, TAGGEDFILE) != EOF) &&
        (numclusts < NUMCLUSTS)) {
  /* determine closest cluster */
  vec.threeprev = threeprev;
  vec.prev = prev;
  vec.curr = curr;
  vec.tag = curr.tag;
  vec.next = next;
  vec.threenext = threenext;
  vec.lastcall = lastcall;
  vec.timetocall = time - timeofcall;
  vec.lastcallval = lastcallval;
  vec.normalize();
  if (time == 2) {
    for (i = 0; i < NUMCLUSTS; i++) {
      /* this is a bit of a kludge - set all clusters same so
         that can use existing cluster functions */
      clust[i]->average = vec;
    }
    numclusts++;
    fprintf(stderr, "Time %4d, adding prototype for %c
            time, inttocall(vec.tag));
  switch (vec.tag) {
  case -1:
     xclusts++;
     break;
  case 0:
     cclusts++;
     break;
  case 1:
     aclusts++;
     break;
  case 2:
     gclusts++;
     break;
  case 3:
     tclusts++;
     break;
  }
}
else {
  call = vec.call(clust);
  if (call != vec.tag) {
     /* don't add as prototype if one point short of call that was made */
     if ((!(iscall(vec.tag))) && (call == next.tag)) {
       fprintf(stderr, "%c!", inttocall(vec.tag));
       shiftflag = 1;
     }
     else {
       clust[numclusts++]->average = vec;
       fprintf(stderr, "\nTime %4d, adding prototype for %c [%c]    :   ",
                time, inttocall(vec.tag), inttocall(call));
       switch (vec.tag) {
       case -1:
         xclusts++;
         break;
       case 0:
         cclusts++;
         break;
       case 1:
         aclusts++;
         break;
       case 2:
         gclusts++;
         break;
       case 3:
         tclusts++;
         break;
```

```
          }
          if (iscall(call) && iscall(vec.tag)) callmistakes++;
        }
      }
      else fprintf(stderr, "%c", inttocall(vec.tag));
    }

    /* move to next time */
    if (iscall(vec.tag)) {
      lastcall = vec.tag;
      lastcallval = curr.fluorescence(call);
      timeofcall = time;
    }
    time++;
    threeprev = twoprev;
    twoprev = prev;
    prev = curr;
    curr = next;
    next = twonext;
    twonext = threenext;
  }

  fprintf(stderr, "\n");
  if (numclusts < NUMCLUSTS)
    fprintf(stderr, "Finished %d time units with %d prototypes\n",
             time, numclusts);
  else
    fprintf(stderr, "Ran over %d prototype limit at %d time units\n",
             numclusts, time);
  fprintf(stderr,
          "Prototypes:\n  C: %6d\n  T: %6d\n  A: %6d\n  G: %6d\n  X: %6d\n",
          cclusts, tclusts, aclusts, gclusts, xclusts);
  fprintf(stderr, "%d calls incorrectly called as other calls\n",
          callmistakes);
  fclose(datafile);

  /* reopen to beginning */
  datafile = fopen(argv[1], "r");

  /* initialize threeprev through twonext to first six time points */
  threeprev.input(datafile, TAGGEDFILE);
  twoprev.input(datafile, TAGGEDFILE);
  prev.input(datafile, TAGGEDFILE);
  curr.input(datafile, TAGGEDFILE);
  next.input(datafile, TAGGEDFILE);
  twonext.input(datafile, TAGGEDFILE);
```

```
time = 2;
lastcall = 0;
lastcallval = 1;
timeofcall = 0;

/* read through */
fprintf(stderr, "\nCalling data.\n\n");

int prevtag = -1;

while ((threenext.input(datafile, TAGGEDFILE) != EOF)) {
  /* determine closest cluster */
  vec.threeprev = threeprev;
  vec.prev = prev;
  vec.curr = curr;
  vec.tag = curr.tag;
  vec.next = next;
  vec.threenext = threenext;
  vec.lastcall = lastcall;
  vec.timetocall = time - timeofcall;
  vec.lastcallval = lastcallval;
  vec.normalize();
  call = vec.call(clust);
  if (iscall(call) && (call != prevtag)) {
    printf("%c", inttocall(call));
    basenum++;
    if (!(basenum%10)) printf(" ");
    if (!(basenum%50)) printf("\n");
    prevtag = call;
  }
  else prevtag = -1;

  /* move to next time */
  if (iscall(vec.tag)) {
    lastcall = vec.tag;
    lastcallval = curr.fluorescence(call);
    timeofcall = time;
  }
  time++;
  threeprev = twoprev;
  twoprev = prev;
  prev = curr;
  curr = next;
  next = twonext;
  twonext = threenext;
}
```

```
    fclose(datafile);
    printf("\n");
    exit(0);
  }
```

```
#include "prototype.h"

#define CALLS_ONLY 0

extern double currmin;

int main(int argc, char *argv[]) {
//  malloc_debug(8);

  int time;          /* current time */
  int right=0, wrong=0;
  int basenum=0;
  int timeofcall;    /* time of last base call */
  int lastcall;      /* last base call */
  int i;
  int call;
  int protnum;
  char ctag[2];      /* tag character */

  if (argc < 3) {
    fprintf(stderr,"Usage: %s <data file> <prototype file>\n", argv[0]);
    exit(1);
  }

  FILE *datafile, *protfile;

  if (!(protfile = fopen(argv[2], "r"))) {
    fprintf(stderr, "%s: Can not open file %s.\n", argv[0], argv[2]);
    exit(1);
  }

  datapoint prev, curr, next; /* previous, current, and next data points */
  vector vec;                 /* current vector */
  vector prot[NUMPROT];       /* prototypes */
  double movefactor;          /* factor for prototype movement */

  /* initialize prototypes */
  for (i = 0; i < NUMPROT; i++) {
    fscanf(protfile, "%lf%lf%lf%lf%1s", &prot[i].prev.c, &prot[i].prev.t,
           &prot[i].prev.a, &prot[i].prev.g, ctag);
    prot[i].prev.tag = calltoint(ctag[0]);
    fscanf(protfile, "%lf%lf%lf%lf%1s", &prot[i].curr.c, &prot[i].curr.t,
           &prot[i].curr.a, &prot[i].curr.g, ctag);
    prot[i].curr.tag = calltoint(ctag[0]);
    fscanf(protfile, "%lf%lf%lf%lf%1s", &prot[i].next.c, &prot[i].next.t,
           &prot[i].next.a, &prot[i].next.g, ctag);
```

```
     prot[i].next.tag = calltoint(ctag[0]);
     prot[i].normalize();
     prot[i].tag = prot[i].curr.tag;
 //    prot[i].print();
   }

   /* read through data file, using calls to update prototypes */

   for (movefactor = 0.2; movefactor >= 0.01; movefactor /= 2.0) {
     fprintf(stderr, "Move factor = %.4f\n" ,movefactor);

     if (!(datafile = fopen(argv[1], "r"))) {
       fprintf(stderr, "%s: Can not (re)open file %s.\n", argv[0], argv[1]);
       exit(1);
     }

     /* initialize prev and curr to first two time points */
     fscanf(datafile, "%lf%lf%lf%lf%ls",
             &prev.c, &prev.t, &prev.a, &prev.g, ctag);
     prev.tag = calltoint(ctag[0]);
     fscanf(datafile, "%lf%lf%lf%lf%ls",
             &curr.c, &curr.t, &curr.a, &curr.g, ctag);
     curr.tag = calltoint(ctag[0]);
     time = 2;
     lastcall = 3;
     timeofcall = 0;

//     fprintf(stderr, "Reading file.\n    Scanning line ");
     while (fscanf(datafile, "%lf%lf%lf%lf%ls",
                    &next.c, &next.t, &next.a, &next.g, ctag) != EOF) {
       next.tag = calltoint(ctag[0]);
       if ((time%500) == 0) {
//        fprintf(stderr, "%d...", time);
       }
       if (iscall(curr.tag)) {
          /* if call, create and normalize vector, find and update closest
             prototype */
          vec.prev = prev;
          vec.curr = curr;
          vec.next = next;
          vec.lastcall = lastcall;
          vec.timetocall = time - timeofcall;
          vec.normalize();
          protnum = vec.call(prot);
          call = prot[protnum].tag;
```

```
        /* determine if correct and move prototype accordingly */
        if (call == curr.tag) {
          /* fprintf(stderr, "Moving prototype %d\n", protnum);
             prot[protnum].print();
             fprintf(stderr, "towards (%.2f)\n", movefactor);
             vec.print(); */
          prot[protnum].movetowards(&vec, movefactor);
          /*      fprintf(stderr, "Result is\n");
                  prot[protnum].print();
                  */
        }
        else {
          fprintf(stderr, "   Moving prototype %d\n", protnum);
          /*      prot[protnum].print();
                  fprintf(stderr, "away from (%.2f)\n", movefactor);
                  vec.print(); */
          prot[protnum].moveaway(&vec, movefactor);
          /*      fprintf(stderr, "Result is\n");
                  prot[protnum].print();
                  */
        }

       basenum++;
       lastcall = curr.tag;
       timeofcall = time;
    }
    /* move to next time */
    time++;
    prev = curr;
    curr = next;
  }

  /*  fprintf(stderr, "New prototypes:\n");
       for (i=0; i < NUMPROT; i++) {
       prot[i].print();
       }
       */
  fclose(datafile);
}

/* now use prototypes to call */
datafile = fopen(argv[1], "r");     /* reopen to beginning of file */
time = 0;
basenum = 0;

double twoprevmin = 0, prevmin = 0;
```

```
int prevcall = -1, prevbase = 0;
int disttocall = 0;

/* initialize prev to first time points */
fscanf(datafile, "%lf%lf%lf%lf%1s",
        &prev.c, &prev.t, &prev.a, &prev.g, ctag);
prev.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%1s",
        &curr.c, &curr.t, &curr.a, &curr.g, ctag);
curr.tag = calltoint(ctag[0]);
time = 2;
lastcall = 0;
timeofcall = 0;

/* read through */
fprintf(stderr, "Calling data.\n\n");
while ((fscanf(datafile, "%lf%lf%lf%lf%1s",
                &next.c, &next.t, &next.a, &next.g, ctag)) != EOF) {
  next.tag = calltoint(ctag[0]);

  /* determine closest prototype */
  vec.prev = prev;
  vec.curr = curr;
  vec.next = next;
  vec.tag = curr.tag;
  vec.lastcall = lastcall;
  vec.timetocall = time - timeofcall;
  vec.normalize();
  protnum = vec.call(prot);
  call = prot[protnum].tag;

  if (CALLS_ONLY) {
    if (iscall(curr.tag)) {
       printf("%c", inttocall(call));

       if (iscall(vec.tag)) {
         lastcall = vec.tag;
         timeofcall = time;
       }

       if (iscall(call)) {
         basenum++;
         lastcall = call;
         timeofcall = time;
         if (call == vec.tag) {
           right++;
```

```
          }
          else {
            wrong++;
            printf("(%c)", inttocall(vec.tag));
          }
        }

        if (basenum%10 == 5) printf(" ");
        if (basenum%50 == 25) printf("\n");
      }
    }

    if (!CALLS_ONLY) {
      /* determine if distance is a local minimum */
      if ((twoprevmin > prevmin) && (prevmin <= currmin)
        && (disttocall > (0.45*avetimetocall(prevcall, prevbase)))) {
/*        ) { */
        lastcall = prevcall;
        timeofcall = time-1;
        printf("%c", inttocall(prevcall));
        basenum++;
        if (!(basenum%79)-) printf("\n");
        disttocall = 0;
        prevbase = prevcall;
      }
      else {
//      printf(".");
        disttocall++;
      }

//     printf(" [%c]", inttocall(prev.tag));
    }

    /* move to next time */
    time++;
    prev = curr;
    curr = next;
    prevcall = call;
    twoprevmin = prevmin;
    prevmin = currmin;
  }

  printf("\n");

  if (CALLS_ONLY) {
    fprintf(stderr, "\nCalling completed.  Read %d time units, %d calls\n",
                time, basenum);
    fprintf(stderr, "Calls right:   %d\nCalls wrong:   %d\nSuccess rate:
%4.3f\n",
                right, wrong, double(right)/double(basenum));
  }

  fclose(datafile);
  exit(0);
}
```

```
/* Contains member functions for cluster and vector classes, as well as
   other auxilliary functions. */

#include "cluster.h"

extern double currmin;
extern int currcluster;

int round(double x) {
  return ((x - int(x) > .5) ? (int(x)+1) : int(x));
}

void datapoint::print(FILE *fp, int flag) {
  /* if flag is one, print tag; if zero don't print tag */
  fprintf(fp, "%10.6f ", c);
  fprintf(fp, "%10.6f ", t);
  fprintf(fp, "%10.6f ", a);
  fprintf(fp, "%10.6f", g);
  if (flag) fprintf(fp, "%6c", inttocall(tag));
  fprintf(fp, "\n");
}

void vector::print(FILE *fp) {
  threeprev.print(fp);
  prev.print(fp);
  curr.print(fp);
  next.print(fp);
  threenext.print(fp);
  fprintf(fp, "%3d %3d %10.6f %10.6f %10.6f %10.6f %3d\n",
          lastcall, timetocall, lastcallval, max, threeprevmax, threenextmax,
          tag);
}

int datapoint::input(FILE *fp, int flag) {
  /* read in from file.  If flag is 0, don't look for tag. Sets all negative
     values to 0. If flag is 1, looks for alphabetic tags, if 2, looks for
     numeric tags. */
  int rv;
  char ctag[2];

  rv = fscanf(fp, "%lf %lf %lf %lf", &c, &t, &a, &g);
  if ((flag == 1) && (rv != EOF)) {
    rv = fscanf(fp, "%1s", ctag);
    tag = calltoint(ctag[0]);
  }
  if ((flag == 2) && (rv != EOF)) rv = fscanf(fp, "%d", &tag);
  if (c < 0) c = 0;
  if (t < 0) t = 0;
  if (a < 0) a = 0;
  if (g < 0) g = 0;

  return rv;
}

double datapoint::fluorescence(int tag) {
  switch (tag) {
  case 0:
    return c;
    break;
  case 1:
    return t;
    break;
  case 2:
    return a;
    break;
  case 3:
    return g;
    break;
  default:
    return 0;
    break;
  }
}

void vector::input(FILE *fp) {
  /* read in from file */
  threeprev.input(fp);
  prev.input(fp);
  curr.input(fp);
  next.input(fp);
  threenext.input(fp);
  fscanf(fp, "%d %d %lf %lf %lf %lf %d\n",
          &lastcall, &timetocall, &lastcallval,
          &max, &threeprevmax, &threenextmax, &tag);
}

void vector::normalize() {
  /* normalize all fluorescence values so that max value is 100, and set
     max field to absolute max.  */

  double max1, max2, max3, max4, max5, max6, max7, max8, max9, max10;
```

```
/* quick hack! */

/* round one - compare pairs of values */
max1 = (prev.c > prev.a)? prev.c : prev.a;
max2 = (prev.g > prev.t)? prev.g : prev.t;
max3 = (curr.c > curr.a)? curr.c : curr.a;
max4 = (curr.g > curr.t)? curr.g : curr.t;
max5 = (next.c > next.a)? next.c : next.a;
max6 = (next.g > next.t)? next.g : next.t;
max7 = (threeprev.c > threeprev.a)? threeprev.c : threeprev.a;
max8 = (threeprev.g > threeprev.t)? threeprev.g : threeprev.t;
max9 = (threenext.c > threenext.a)? threenext.c : threenext.a;
max10 = (threenext.g > threenext.t)? threenext.g : threenext.t;

/* round two - compare winners of round 1 */
max1 = (max1 > max2)? max1 : max2;
max2 = (max3 > max4)? max3 : max4;
max3 = (max5 > max6)? max5 : max6;
max4 = (max7 > max8)? max7 : max8;
threeprevmax = max4;
max5 = (max9 > max10)? max9 : max10;
threenextmax = max5;

/* round three - determine largest winner of round 2 */
max1 = (max1 > max2)? max1 : max2;
max2 = (max3 > max4)? max3 : max4;
max3 = max5;

/* round four - determine largest winner of round 3 */
max1 = (max1 > max2)? max1 : max2;
max1 = (max1 > max3)? max1 : max3;
max = max1; /* set max field */

/* max1 is now max.  Normalize values */
if (max1 != 0) {
  threeprev.c = threeprev.c / max1;
  threeprev.a = threeprev.a / max1;
  threeprev.g = threeprev.g / max1;
  threeprev.t = threeprev.t / max1;
  prev.c = prev.c / max1;
  prev.a = prev.a / max1;
  prev.g = prev.g / max1;
  prev.t = prev.t / max1;
  curr.c = curr.c / max1;
  curr.a = curr.a / max1;
  curr.g = curr.g / max1;
```

```
    curr.t = curr.t / max1;
    next.c = next.c / max1;
    next.a = next.a / max1;
    next.g = next.g / max1;
    next.t = next.t / max1;
    threenext.c = threenext.c / max1;
    threenext.a = threenext.a / max1;
    threenext.g = threenext.g / max1;
    threenext.t = threenext.t / max1;
  }
}

int iscall(int call) {
  /* determines whether a data line has been called */
  return((call >= 0) && (call <= 4));
}

int calltoint(char call) {
  /* converts character tag to int */
  switch (call) {
  case 'C':
    return 0;
  case 'A':
    return 1;
  case 'G':
    return 2;
  case 'T':
    return 3;
  case 'X':
  default:
    return -1;
  }
}

char inttocall(int tag) {
  switch (tag) {
  case 0:
    return 'C';
  case 1:
    return 'A';
  case 2:
    return 'G';
  case 3:
    return 'T';
  case -1:
  default:
    return 'X';
  }
}

void cluster:: addvec(vector *vec) {
  /* adds the vector vec to the cluster clust */
  if (vec->max != 0) {
    if (size >= 64) {
      fprintf(stderr, "Warning: Too many vectors in cluster %c %c\n",
               inttocall(vec->lastcall),inttocall(vec->tag));
    }
    else {
      data[size] = *vec;
      size++;
    }
  }
}

void cluster:: create_average() {
  /* creates arithmetic average vector from vectors in data array */
  int i;
  double tpcs = 0.0, tpas = 0.0, tpgs = 0.0, tpts = 0.0;
  double pcs = 0.0, pas = 0.0, pgs = 0.0, pts = 0.0;
  double ccs = 0.0, cas = 0.0, cgs = 0.0, cts = 0.0;
  double ncs = 0.0, nas = 0.0, ngs = 0.0, nts = 0.0;
  double tncs = 0.0, tnas = 0.0, tngs = 0.0, tnts = 0.0;
  double ttcs = 0.0, lcvs = 0.0, ms = 0.0, pms = 0.0, nms = 0.0;
  vector *cp;

  /* sum up each vector component */
  if (size == 0) fprintf(stderr, "No data points in this cluster!\n");

//  fprintf(stderr, "\nCluster %c%c:\n    ", inttocall(data[0].lastcall),
//         inttocall(data[0].tag));

  for (i = 0; i < size; i++) {
    cp = &data[i];
//    if (i>0) fprintf(stderr, "%d ", cp->timetocall);
    tpcs += cp->threeprev.c;
    tpas += cp->threeprev.a;
    tpgs += cp->threeprev.g;
    tpts += cp->threeprev.t;
    pcs += cp->prev.c;
    pas += cp->prev.a;
    pgs += cp->prev.g;
    pts += cp->prev.t;
  ccs += cp->curr.c;
  cas += cp->curr.a;
  cgs += cp->curr.g;
  cts += cp->curr.t;
  ncs += cp->next.c;
  nas += cp->next.a;
  ngs += cp->next.g;
  nts += cp->next.t;
  tncs += cp->threenext.c;
  tnas += cp->threenext.a;
  tngs += cp->threenext.g;
  tnts += cp->threenext.t;
  if (i>0) {  /* ignore timetocall and lastcallval for first data point,
                 since may be bad */
    ttcs += cp->timetocall;
    lcvs += cp->lastcallval;
  }
  ms += cp->max;
  pms += cp->threeprevmax;
  nms += cp->threenextmax;
}

/* take averages */
average.threeprev.c = tpcs/size;
average.threeprev.a = tpas/size;
average.threeprev.g = tpgs/size;
average.threeprev.t = tpts/size;
average.prev.c = pcs/size;
average.prev.a = pas/size;
average.prev.g = pgs/size;
average.prev.t = pts/size;
average.curr.c = ccs/size;
average.curr.a = cas/size;
average.curr.g = cgs/size;
average.curr.t = cts/size;
average.next.c = ncs/size;
average.next.a = nas/size;
average.next.g = ngs/size;
average.next.t = nts/size;
average.threenext.c = tncs/size;
average.threenext.a = tnas/size;
average.threenext.g = tngs/size;
average.threenext.t = tnts/size;
average.timetocall = round(ttcs/(size-1));
average.lastcallval = lcvs/(size-1);
average.max = ms/size;
```

```
  average.threeprevmax = pms/size;
  average.threenextmax = nms/size;

  average.lastcall = data[0].lastcall;   /* same for whole cluster */
  average.tag = data[0].curr.tag;        /* same for whole cluster */

//  fprintf(stderr, "(average %.2f %d)", ttcs/(size-1), average.timetocall);

}

double cluster:: distance(vector *vec) {
  double dist, vmax, amax, vlast, alast;

  if (vec->max <= 0) vmax = 1;
  else vmax = vec->max;
  if (average.max <= 0) amax = 1;
  else amax = average.max;
  if (vec->lastcallval <= 0) vlast = 1;
  else vlast = vec->lastcallval;
  if (average.lastcallval <= 0) alast = 1;
  else alast = average.lastcallval;

  dist =((
//         pow((vec->threeprev.c - average.threeprev.c), 2) +
//         pow((vec->threeprev.a - average.threeprev.a), 2) +
//         pow((vec->threeprev.g - average.threeprev.g), 2) +
//         pow((vec->threeprev.t - average.threeprev.t), 2) +
           pow((vec->prev.c - average.prev.c)*1.5, 2) +
           pow((vec->prev.a - average.prev.a)*1.5, 2) +
           pow((vec->prev.g - average.prev.g)*1.5, 2) +
           pow((vec->prev.t - average.prev.t)*1.5, 2) +
           pow((vec->curr.c - average.curr.c)*2.0, 2) +
           pow((vec->curr.a - average.curr.a)*2.0, 2) +
           pow((vec->curr.g - average.curr.g)*2.0, 2) +
           pow((vec->curr.t - average.curr.t)*2.0, 2) +
           pow((vec->next.c - average.next.c)*1.5, 2) +
           pow((vec->next.a - average.next.a)*1.5, 2) +
           pow((vec->next.g - average.next.g)*1.5, 2) +
           pow((vec->next.t - average.next.t)*1.5, 2) +
           pow(vec->prev.c - average.prev.c + vec->next.c - average.next.c, 2) +
           pow(vec->prev.a - average.prev.a + vec->next.t - average.next.t, 2) +
           pow(vec->prev.g - average.prev.g + vec->next.a - average.next.a, 2) +
           pow(vec->prev.t - average.prev.t + vec->next.g - average.next.g, 2) +
//         pow((vec->threenext.c - average.threenext.c), 2) +
//         pow((vec->threenext.a - average.threenext.a), 2) +
//         pow((vec->threenext.g - average.threenext.g), 2) +
//        pow((vec->threenext.t - average.threenext.t), 2) +
          pow(vec->threeprev.c - average.threeprev.c +
              vec->threenext.c - average.threenext.c, 2) +
          pow(vec->threeprev.a - average.threeprev.a +
              vec->threenext.t - average.threenext.t, 2) +
          pow(vec->threeprev.g - average.threeprev.g +
              vec->threenext.a - average.threenext.a, 2) +
          pow(vec->threeprev.t - average.threeprev.t +
              vec->threenext.g - average.threenext.g, 2)) *
         (pow(0.3*(vec->timetocall - average.timetocall), 2) + 1) +
//       (pow(0.01*(vmax/vlast - amax/alast), 2) + 1)
//       ((vmax < amax)? (pow(0.3*(vmax - amax), 2) + 1) : 1) +
         (pow((log(vmax) - log(amax))*.5, 2) +
          pow((log(vec->threeprevmax) - log(average.threeprevmax))*.25, 2) +
          pow((log(vec->threenextmax) - log(average.threenextmax))*.25, 2))
         );


//  if (vec->lastcall == average.lastcall) dist = dist-1000;

  return(dist);
}

int vector::call(cluster **clust) {
  /* determines tag of closest cluster */

  double dist[NUMCLUSTS];
  int i;

  for (i=0; i < NUMCLUSTS; i++) {
    dist[i] = clust[i]->distance(this);
//    clust[lastcall*4 + i]->distance(this);
  }

  /* note: for coding purposes, use linear min.  Should change
     to log time min by pairing */
  double min;

  /* go through array updating min as you go */
  min = dist[0];
  for (i = 1; i < NUMCLUSTS; i++) {
    if (dist[i] < min) min = dist[i];
  }

  /* min is now minimum distance - determine cluster */
 //  if (min < 10000) {
  /* set currmin to this value */
  currmin = min;
//  printf("\n%15.0f ", min);
  for (i = 0; i < NUMCLUSTS; i++) {
    if (min == dist[i]) {
/*      printf("Current vector: \n");
      this->print();
      printf("Closest cluster: \n");
      clust[i]->average.print();   */
//      printf(" (%d) ", i);
      currcluster = i;
      return (clust[i]->average.tag);
    }
  }
//}

  /* default */
  return(-1);
}

int vector::findclust() {
  /* returns index of cluster to which this vector belongs */
  return(4*lastcall + curr.tag);
}

double avedist(int prev, int curr, cluster **clust) {
  /* find cluster and return average timetocall */
  return(clust[4*prev + curr]->average.timetocall);
}

int clusterprevtag(int cl) {
  /* return tag of previous base of cluster number cl */
  return (cl/4);
}

int clustercurrtag(int cl) {
  /* return tag of current base of cluster number cl */
  return (cl%4);
}
```

```
/* Adds consecutive rows in a four channel file.  Reads from standard input,
   writes to standard output. */

extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
}

main() {

  double p1, r1, y1, g1, p2, r2, y2, g2;

  while ((scanf("%lf %lf %lf %lf", &p1, &r1, &y1, &g1) == 4) &&
          (scanf("%lf %lf %lf %lf", &p2, &r2, &y2, &g2) == 4))
    /* if odd number of frames, ignores last frame */
    printf("%10.6f %10.6f %10.6f %10.6f\n",
             p1+p2, r1+r2, y1+y2, g1+g2);

  exit(0);
 }
```

```
/* Read two cluster files, produce output that averages corresponding cluster
   values from the two clusters. */

#include "cluster.h"

int main(int argc, char *argv[]) {

  FILE *clusterfile1, *clusterfile2;

  if (argc != 3) {
    fprintf(stderr, "Usage: %s <cluster file> <cluster file>\n", argv[0]);
    exit(1);
  }

  if (!(clusterfile1 = fopen(argv[1], "r"))) {
    fprintf(stderr, "Can not open cluster file %s.\n", argv[1]);
    exit(1);
  }

  if (!(clusterfile2 = fopen(argv[2], "r"))) {
    fprintf(stderr, "Can not open cluster file %s.\n", argv[2]);
    exit(1);
  }

  int i;
  vector vec1, vec2, average;

  for (i = 0; i < NUMCLUSTS; i++) {
    vec1.input(clusterfile1);
    vec2.input(clusterfile2);
    averagetwovecs(&vec1, &vec2, &average);
    average.print();
  }
}
```

```
#include "cluster.h"

/* Using all calls, create "clusters" for each base called
   (plus additional information if desired).
   Find average vector of each cluster.
   Run through file again, this time calling even numbered calls by
   closest cluster.
   Possible tags are C, A, G, T, (N?), and X, where X is for no base.
*/

int main(int argc, char *argv[]) {
  int time;           /* current time */
  int basenum=0;
  int timeofcall;     /* time of last base call */
  int lastcall;       /* last base call */
  int i;
  char ctag[2];       /* for reading tag character */
  if (argc < 2) {
    fprintf(stderr, "Usage: %s <datafile>\n", argv[0]);
    exit(1);
  }

  FILE *datafile;

  if (!(datafile = fopen(argv[1], "r"))) {
    fprintf(stderr, "Can not open data file %s.\n", argv[1]);
    exit(1);
  }

  datapoint threeprev, twoprev, twonext, threenext;
  datapoint prev, curr, next; /* previous, current, and next data points */
  vector vec;                 /* current vector */
  cluster *clust[NUMCLUSTS];  /* clusters */

  cluster *rv;

  /* read through, collecting calls */
  for (i = 0; i < NUMCLUSTS; i++) {  /* initialize */
    if ((rv = (cluster *) malloc(sizeof(cluster))) != NULL) {
      clust[i] = rv;
      clust[i]->size = 0;
    }
    else {
      fprintf(stderr, "\nNot enough memory - Cluster %d\n", i);
      exit(1);
    }
```

```
}

/* initialize threeprev through twonext to first six time points */
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &threeprev.c, &threeprev.t, &threeprev.a, &threeprev.g, ctag);
threeprev.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &twoprev.c, &twoprev.t, &twoprev.a, &twoprev.g, ctag);
twoprev.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &prev.c, &prev.t, &prev.a, &prev.g, ctag);
prev.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &curr.c, &curr.t, &curr.a, &curr.g, ctag);
curr.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &next.c, &next.t, &next.a, &next.g, ctag);
next.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &twonext.c, &twonext.t, &twonext.a, &twonext.g, ctag);
twonext.tag = calltoint(ctag[0]);
time = 2;
lastcall = 3;
timeofcall = 0;

fprintf(stderr, "Reading file.\n    Scanning line ");
while ((fscanf(datafile, "%lf%lf%lf%lf%ls",
                 &threenext.c, &threenext.t, &threenext.a, &threenext.g, ctag))
        != EOF) {
  threenext.tag = calltoint(ctag[0]);
  if ((time%500) == 0) {
    fprintf(stderr, "%d...", time);
  }
  if (iscall(curr.tag)) {
    /* if call, create and normalize vector, and
        add to appropriate cluster */
    vec.threeprev = threeprev;
    vec.prev = prev;
    vec.curr = curr;
    vec.next = next;
    vec.threenext = threenext;
    vec.tag = curr.tag;
    vec.lastcall = lastcall;
    vec.timetocall = time - timeofcall;
    vec.normalize();
    clust[vec.findclust()]->addvec(&vec);
```

```
      basenum++;
      lastcall = curr.tag;
      timeofcall = time;
    }
    /* move to next time */
    time++;
    threeprev = twoprev;
    twoprev = prev;
    prev = curr;
    curr = next;
    next = twonext;
    twonext = threenext;
  }

  fprintf(stderr, "\nFile scanned.  Creating clusters ...");

  /* create cluster averages */
  fprintf(stderr, "\n");

  for (i = 0; i < NUMCLUSTS; i++) {
    clust[i]->create_average();
  }

  fprintf(stderr, "\nClustering completed.  Read %d time units, ", time);
  fprintf(stderr, "%d total calls\n", basenum);
/*  for (i = 0; i < NUMCLUSTS; i++) {
    fprintf(stderr, "\nCluster %d, size %d.\n", i, clust[i]->size);
    fprintf(stderr, "Average vector:\n");
    clust[i]->average.print();
  }*/
  fprintf(stderr, "\n");

  /* reset */
  fclose(datafile);
  datafile = fopen(argv[1], "r");    /* reopen to beginning of file */
  time = 0;
  basenum = 0;

  char call;
  int right = 0, wrong = 0;

  /* initialize threeprev through twonext to first six time points */
  fscanf(datafile, "%lf%lf%lf%lf%1s",
          &threeprev.c, &threeprev.t, &threeprev.a, &threeprev.g, ctag);
  threeprev.tag = calltoint(ctag[0]);
  fscanf(datafile, "%lf%lf%lf%lf%1s",
        &twoprev.c, &twoprev.t, &twoprev.a, &twoprev.g, ctag);
twoprev.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &prev.c, &prev.t, &prev.a, &prev.g, ctag);
prev.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &curr.c, &curr.t, &curr.a, &curr.g, ctag);
curr.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &next.c, &next.t, &next.a, &next.g, ctag);
next.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &twonext.c, &twonext.t, &twonext.a, &twonext.g, ctag);
twonext.tag = calltoint(ctag[0]);
time = 2;
lastcall = 0;
timeofcall = 0;

/* read through */
fprintf(stderr, "Calling data.\n\n");
while ((fscanf(datafile, "%lf%lf%lf%lf%ls",
                 &threenext.c, &threenext.t, &threenext.a, &threenext.g, ctag))
        != EOF) {
  threenext.tag = calltoint(ctag[0]);
  if (iscall(curr.tag)) {
    /* determine call */
    vec.threeprev = threeprev;
    vec.prev = prev;
    vec.curr = curr;
    vec.next = next;
    vec.threenext = threenext;
    vec.tag = curr.tag;
    vec.lastcall = lastcall;
    vec.timetocall = time - timeofcall;
    vec.normalize();
    call = vec.call(clust);
    printf("%c", inttocall(call));

    if (iscall(vec.tag)) {
       lastcall = vec.tag;
       timeofcall = time;
    }

    if (iscall(call)) {
       basenum++;
       lastcall = call;
```

```
        timeofcall = time;
        if (call == vec.tag) {
          right++;
        }
        else {
          wrong++;
          printf("(%c)", inttocall(vec.tag));
        }
      }

      if (basenum%10 == 5) printf(" ");
      if (basenum%50 == 25) printf("\n");
    }

    /* move to next time */
    time++;
    threeprev = twoprev;
    twoprev = prev;
    prev = curr;
    curr = next;
    next = twonext;
    twonext = threenext;
  }

  printf("\n");

  fprintf(stderr, "\nCalling completed.  Read %d time units, %d calls\n",
          time, basenum);
  fprintf(stderr,
          "Calls right:   %d\nCalls wrong:   %d\nSuccess rate:  %4.3f\n",
          right, wrong, double(right)/double(basenum));

  fclose(datafile);
  exit(0);
}
```

```
/* Create clusters from argument file if there is one, standard input
   otherwise.  Write cluster averages to standard output. */

#include "cluster.h"

/* Using all calls, create "clusters" for each base called
   (plus additional information if desired).
   Find average vector of each cluster.
*/

int main(int argc, char *argv[]) {
  int time;          /* current time */
  int basenum=0;
  int timeofcall;    /* time of last base call */
  int lastcall;      /* last base call */
  double lastcallval; /* fluorescence value of last call */
  int i;

  FILE *datafile;

  if (argc == 1) datafile = stdin;
  else {
    if (!(datafile = fopen(argv[1], "r"))) {
      fprintf(stderr, "Can not open data file %s.\n", argv[1]);
      exit(1);
    }
  }

  datapoint threeprev, twoprev, twonext, threenext;
  datapoint prev, curr, next; /* previous, current, and next data points */
  vector vec;                 /* current vector */
  cluster *clust[NUMCLUSTS];  /* clusters */

  cluster *rv;

  /* read through, collecting calls */
  for (i = 0; i < NUMCLUSTS; i++) {  /* initialize */
    if ((rv = (cluster *) malloc(sizeof(cluster))) != NULL) {
      clust[i] = rv;
      clust[i]->size = 0;
    }
    else {
      fprintf(stderr, "\nNot enough memory - Cluster %d\n", i);
      exit(1);
    }
  }
```

```
/* initialize threeprev through twonext to first six time points */
threeprev.input(datafile, 1);
twoprev.input(datafile, 1);
prev.input(datafile, 1);
curr.input(datafile, 1);
next.input(datafile, 1);
twonext.input(datafile, 1);
time = 2;
lastcall = 3;
timeofcall = 0;
lastcallval = 1.0;

fprintf(stderr, "Reading file.\n    Scanning line ");
while (threenext.input(datafile, 1) != EOF) {
  if ((time%500) == 0) {
    fprintf(stderr, "%d...", time);
  }
  if (iscall(curr.tag)) {
    /* if call, create and normalize vector, and
        add to appropriate cluster */
    vec.threeprev = threeprev;
    vec.twoprev = twoprev;
    vec.prev = prev;
    vec.curr = curr;
    vec.next = next;
    vec.twonext = twonext;
    vec.threenext = threenext;
    vec.tag = curr.tag;
    vec.lastcall = lastcall;
    vec.timetocall = time - timeofcall;
    vec.lastcallval = lastcallval;
    /* update lastcallval before normalizing */
    lastcallval = curr.fluorescence(curr.tag);
    vec.normalize();
    clust[vec.findclust()]->addvec(&vec);
    basenum++;
    lastcall = curr.tag;
    timeofcall = time;
  }
  /* move to next time */
  time++;
  threeprev = twoprev;
  twoprev = prev;
  prev = curr;
  curr = next;
```

```
    next = twonext;
    twonext = threenext;
  }

  fprintf(stderr, "\nFile scanned.  Creating clusters ...");

  /* create cluster averages */
  fprintf(stderr, "\n");

  for (i = 0; i < NUMCLUSTS; i++) {
    clust[i]->create_average();
  }

  fprintf(stderr, "\nClustering completed.  Read %d time units, ", time);
  fprintf(stderr, "%d total calls\n", basenum);
  for (i = 0; i < NUMCLUSTS; i++) {
//    fprintf(stderr, "\nCluster %d, size %d.\n", i, clust[i]->size);
//    fprintf(stderr, "Average vector:\n");
    clust[i]->average.print();
  }
//  fprintf(stderr, "\n");

  fclose(datafile);
  exit(0);
}
```

```
/*  convert.c
 *
 * 4 August, 1993
 * Rebecca N. Wright
 *
 * Program to convert binary raw CCD data into smaller, ASCII files.
 *
 * Assumes input file begins with header as specified in the CSMA file
 * head.doc. NOTE: throughout CSMA documentation, integer means short integer.
 * Currently, only unscrambled data is supported, along with the following
 * data types are supported:
 *
 *    datatype from header:
 *        1 -> long integer (4 byte)
 *        2 -> integer (2 byte)                 (Actually SHORT int)
 *        3 -> unsigned integer (2 byte)        (Actually unsigned SHORT int)
 *
 * First argument is file to be processed.  Optional second argument
 * specifies number of lanes.
 * Prompts for number of lanes (if not specified as command line argument)
 * and super-pixels per lane, and adds together
 * to form lanes.
 *
 * Output is one file for each lane, with two digit
 * lane number appended to filename.  Each output file has four columns
 * (C, T, A, G) and one row for each time (starting with time 0).
 *
 * Exit value: 0 if successful, 1 if error.
 *
 */

#include <stdio.h>
#include <stdlib.h>

#define HEADER_SIZE 4100
#define HEADER_TYPE short int
#define DATA_TYPE_1 long int
#define DATA_TYPE_2 short int
#define DATA_TYPE_3 unsigned short int
#define GENERAL_TYPE long int

#define NOSCAN_LOC 34
#define FACCOUNT_LOC 42
#define DATA_TYPE_LOC 108
```

```
#define STRIPE_LOC 656
#define SCRAMBLE_LOC 658

#define CHUNK_SIZE 1024

struct lane {
  int low;              /* what superpixel does lane begin? */
  int high;             /* what superpixel does lane end? */
  char filename[16];    /* name of output file for lane */
  FILE *fp;             /* pointer to output file for lane */
};

main(int argc, char *argv[]) {
  FILE *infile;
  HEADER_TYPE *headerbuf;
  HEADER_TYPE total_stripes, num_super_pixels, data_type, stripes_per_frame;
  HEADER_TYPE scramble, num_frames, frame_size;
  DATA_TYPE_1 *databuf1;  /* for datatype 1 */
  DATA_TYPE_2 *databuf2;  /* for datatype 2 */
  DATA_TYPE_3 *databuf3;  /* for datatype 3 */
  GENERAL_TYPE sum;
  int i, curr_frame, curr_stripe, curr_lane, curr_val, num_lanes, left_end;
  int in, offset, total_vals, frames_left, frames_read;
  int this_chunk_size, prefixindex;
  struct lane *lanes;
  char prefix[32], *infilename;

  int hs = sizeof(HEADER_TYPE), hn = HEADER_SIZE/hs;

  /* check for correct usage */
  if (argc < 2) {
    fprintf(stderr, "Usage:  %s <binary data file>\n", argv[0]);
    exit(0);
  }

  /* open data file */
  if (!(infile = fopen(argv[1], "r"))) {
    fprintf(stderr, "%s: Can not open %s\n", argv[0], argv[1]);
    exit(1);
  }

  infilename = argv[1];

  /* read in header and extract needed fields */
  if (!(headerbuf = (HEADER_TYPE *) calloc(hn, hs))) {
    fprintf(stderr, "%s: Can not allocate memory for header.\n", argv[0]);
  exit(1);
}
if (fread(headerbuf, hs, hn, infile) != hn) {
  fprintf(stderr, "%s: Incorrect file type %s -  header too short.\n",
          argv[0], argv[1]);
  exit(1);
}

total_stripes = headerbuf[NOSCAN_LOC/sizeof(HEADER_TYPE)];
num_super_pixels = headerbuf[FACCOUNT_LOC/sizeof(HEADER_TYPE)];
data_type = headerbuf[DATA_TYPE_LOC/sizeof(HEADER_TYPE)];
stripes_per_frame = headerbuf[STRIPE_LOC/sizeof(HEADER_TYPE)];
scramble = headerbuf[SCRAMBLE_LOC/sizeof(HEADER_TYPE)];
num_frames = total_stripes / stripes_per_frame;
total_vals = total_stripes * num_super_pixels;
frame_size = stripes_per_frame * num_super_pixels;

printf("\n");
printf("There are %d stripes, %d stripes per frame, %d super pixels.\n",
        total_stripes, stripes_per_frame, num_super_pixels);
printf("Data type is %d.\n", data_type);
printf("Data is %sscrambled.\n", (scramble == 1)? "un" : "");
printf("\n");

if (scramble != 1) {
  fprintf(stderr, "%s: Scrambled data unsupported.\n", argv[0]);
  exit(1);
}

/* prompt for lane information and open output file for each lane */
if (argc > 2) num_lanes = atoi(argv[2]);
else {
  printf("How many lanes are there? (1 - %d) ", num_super_pixels);
  scanf("%d", &num_lanes);
}
if ((num_lanes < 1) || (num_lanes > num_super_pixels)) {
  fprintf(stderr, "%s: Number of lanes out of range.\n", argv[0]);
  exit(1);
}

if (!(lanes = (struct lane *) calloc(num_lanes, sizeof(struct lane)))) {
  fprintf(stderr, "%s: Can not allocate memory for lanes.\n", argv[0]);
  exit(1);
}

/* strip out filename */
```

```
prefixindex = 0;
while ((*infilename != '.') &&
        (*infilename != '\0')) {
  if (*infilename == '/') {
    prefixindex = 0;
    infilename++;
  }
  else {
    prefix[prefixindex++] = *infilename++;
  }
}

left_end = 1;
for (i = 0; i < num_lanes; i++) {
  printf("Low superpixel for lane %d? (%d - %d) ",
          i+1, left_end, num_super_pixels);
  scanf("%d", &in);
  if ((in < left_end) || (in > num_super_pixels)) {
    fprintf(stderr, "%s: Low superpixel out of range.\n", argv[0]);
    exit(1);
  }
  else {
    lanes[i].low = in;
    left_end = in;
  }
  printf("High superpixel for lane %d? (%d - %d) ",
          i+1, left_end, num_super_pixels);
  scanf("%d", &in);
  if ((in < left_end) || (in > num_super_pixels)) {
    fprintf(stderr, "%s: High superpixel out of range.\n", argv[0]);
    exit(1);
  }
  else {
    lanes[i].high = in;
    left_end = in+1;
  }
  /* write i+1 to end of filename prefix */
  sprintf(lanes[i].filename, "%s", prefix);
  if (i < 9) sprintf(lanes[i].filename + prefixindex, "0%d", i+1);
  else sprintf(lanes[i].filename + prefixindex, "%d", i+1);
  printf("File name for lane %d is %s\n", i+1, lanes[i].filename);
  if (!(lanes[i].fp = fopen(lanes[i].filename, "w"))) {
    fprintf(stderr, "%s: Can't open output file %s.\n", argv[0],
             lanes[i].filename);
    exit(1);
  }
```

```
}

/* read in a chunk of frames at a time and output to desired format */
frames_left = num_frames;
frames_read = 0;

switch (data_type) {
  case 1:
    databuf1 = (DATA_TYPE_1 *) calloc(CHUNK_SIZE*frame_size,
                                      sizeof(DATA_TYPE_1));
    break;
  case 2:
    databuf2 = (DATA_TYPE_2 *) calloc(CHUNK_SIZE*frame_size,
                                      sizeof(DATA_TYPE_2));
    break;
  case 3:
    databuf3 = (DATA_TYPE_3 *) calloc(CHUNK_SIZE*frame_size,
                                      sizeof(DATA_TYPE_3));
    break;
  default:
    fprintf(stderr, "%s: Unsupported data type %d.\n", argv[0], data_type);
    exit(1);
  }

printf("Processing frame ");
while (frames_left > 0) {
  this_chunk_size = (frames_left < CHUNK_SIZE)? frames_left : CHUNK_SIZE;

  /* depending on data type, use appropriate data buffer for reading data */
  switch (data_type) {
  case 1:
    if (fread(databuf1, sizeof(DATA_TYPE_1), this_chunk_size*frame_size,
                infile) !=  this_chunk_size*frame_size) {
       fprintf(stderr, "%s: Unexpected end of file %s.\n", argv[0], argv[1]);
       exit(1);
    }
    for (i = 0; i < this_chunk_size*frame_size; i++) {
       printf("%6d\n", databuf1[i]);
    }
    break;
  case 2:
    if (fread(databuf2, sizeof(DATA_TYPE_2), this_chunk_size*frame_size,
                infile) !=  this_chunk_size*frame_size) {
       fprintf(stderr, "%s: Unexpected end of file %s.\n", argv[0], argv[1]);
       exit(1);
    }
```

```
      for (i = 0; i < this_chunk_size*frame_size; i++) {
         printf("%6d\n", databuf2[i]);
      }
      break;
    case 3:
      /* read in chunk of frames */
      if (fread(databuf3, sizeof(DATA_TYPE_3), this_chunk_size*frame_size,
                  infile) !=  this_chunk_size*frame_size) {
         fprintf(stderr, "%s: Unexpected end of file %s.\n", argv[0], argv[1]);
         exit(1);
      }

      /* process frame chunk */
      for (curr_frame = 0; curr_frame < this_chunk_size; curr_frame++) {
         /* process one frame */
         if (!(frames_read%500)) printf("%d...", frames_read+1);
/*       printf("Frame %d\n", frames_read+1);        */
         for (curr_stripe = 0; curr_stripe < stripes_per_frame; curr_stripe++) {
           /* process one stripe - find lanes */
/*         printf(" Stripe %d\n", curr_stripe+1);  */
           for (curr_lane = 0; curr_lane < num_lanes; curr_lane++) {
             /* process one lane - add up entries low to high */
/*           printf("  Lane %d: low is %d, high is %d.\n", curr_lane+1,
                     lanes[curr_lane].low, lanes[curr_lane].high); */
             sum = 0;
             offset = (curr_frame * frame_size) +
               (curr_stripe * num_super_pixels) + lanes[curr_lane].low-1;
             for (curr_val = lanes[curr_lane].low - 1;
                   curr_val < lanes[curr_lane].high; curr_val++) {
/*             printf("   Adding field %d at %d: %d\n", curr_val+1, offset,
                       databuf3[offset]); */
               sum = sum + databuf3[offset];
               offset++;
             }
/*           printf("   Sum for lane %d is %d\n", curr_lane+1, sum); */
             fprintf(lanes[curr_lane].fp, "%10d", sum);
           }
         }
         frames_read++;
         for (i = 0; i < num_lanes; i++) fprintf(lanes[i].fp, "\n");
      }

      break;
    }

    frames_left -= this_chunk_size;
```

```
  }

  printf("Done\n");
  for (i = 0; i < num_lanes; i++) fclose(lanes[i].fp);
  fclose(infile);
  exit(0);
}
```

```
/* Determines edit distance between two sequences of letters as specified
   in two argument files (ignores whitespace in sequences).  If one argument,
   reads sequence1 from standard input, sequence two from argument file.
   Uses a dynamic programming algorithm.  In order to make the final output
   simpler, runs sequences backwards along edges of matrix. */

#define debug 0

extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
}

class entry {
 public:
  int d;      /* distance */
  int p;      /* previous - 0 for none, 1 for left, 2 for diagonal, 3 for up */
};

char *seq1, *seq2;
int length1, length2;

int equal(int i, int j) {
  return (seq1[length1-i] == seq2[length2-j]);
}

int min(int a, int b) {
  return (a < b? a : b);
}

main(int argc, char **argv) {
  FILE *seq1file, *seq2file;
  char c[2];

  entry **distance;  /* edit distance matrix */

//  malloc_debug(8);

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <sequence file> <sequence file>\n", argv[0]);
    exit(1);
  }

  if (argc == 3) {
    /* open sequence files */
    seq1file = fopen(argv[1], "r");
    seq2file = fopen(argv[2], "r");

    if (!seq2file) {
      fprintf(stderr, "Can not open sequence file %s.\n", argv[2]);
      exit(1);
    }
  }

  if (argc == 2) {
    seq1file = stdin;
    seq2file = fopen(argv[1], "r");
  }

  if (!seq1file) {
    fprintf(stderr, "Can not open sequence file %s.\n", argv[1]);
    exit(1);
  }

  int buf1size = 512, buf2size = 512;

  /* read in sequences */
  seq1 = (char *) malloc(buf1size * sizeof(char));
  seq2 = (char *) malloc(buf2size * sizeof(char));

  /* read sequences in, ignoring whitespace */
  length1 = 0;
  while (fscanf(seq1file, "%1s", c) == 1) {
    if (buf1size <= length1) {
      buf1size *= 2;
      seq1 = (char *) realloc(seq1, buf1size*sizeof(char));
    }
    seq1[length1++] = c[0];
  }

  length2 = 0;
  while (fscanf(seq2file, "%1s", c) == 1) {
    if (buf2size <= length2) {
      buf2size *= 2;
      seq2 = (char *) realloc(seq2, buf2size*sizeof(char));
    }
    seq2[length2++] = c[0];
  }

  printf("\nBases: %4d\n       %4d\n", length1, length2);

  /* allocate distance array */
```

```
distance = (entry **) calloc(length1+1, sizeof(entry *));

int i,j;

for (i=0; i <= length1; i++) {
  distance[i] = (entry *) calloc(length2+1, sizeof(entry));
}

entry *curr;

/* initialize (dummy) outer row and column */
curr = &distance[0][0];
curr->d = 0;
curr->p = 0;

for (j = 1; j <= length2; j++) {
  curr = &distance[0][j];
  curr->d = j;
  curr->p = 1;
}

for (i = 1; i <= length1; i++) {
  curr = &distance[i][0];
  curr->d = i;
  curr->p = 3;
}

/* actual entries */
for (i = 1; i <= length1; i++) {
  for (j = 1; j <= length2; j++) {
    curr = &distance[i][j];
    /* first get minimum of upper and left neighbor */
    if (distance[i-1][j].d < distance[i][j-1].d) {
       curr->d = distance[i-1][j].d + 1;
       curr->p = 3;
    }
    else {
       curr->d = distance[i][j-1].d + 1;
       curr->p = 1;
    }

    /* if equal, see if diagonal entry is less, if not equal,
     see if diagonal entry + 1 is less */
    if ((distance[i-1][j-1].d + (equal(i,j)?0:1)) < curr->d) {
       curr->d = distance[i-1][j-1].d + (equal(i,j)?0:1);
       curr->p = 2;
    }
  }
}

if (debug) {
  printf("\n  ");
  for (j = 0; j < length2; j++) {
    printf("%c ", seq2[j]);
  }
  printf("\n");
  for (i = 1; i <= length1; i++) {
    printf("%c ", seq1[length1-i]);
    for (j = 1; j <= length2; j++) {
       printf("%d ", distance[i][j].d);
    }
    printf("\n");
  }
  printf("\n");
}


/* starting at lower right corner of matrix, construct matching */
class match {
public:
  char c1;
  char mid;
  char c2;
};

match *matching;
matching = (struct match *) calloc(length1 + length2, sizeof(struct match));

int curr1 = length1, curr2 = length2, matchlength = 0;
match *currmatch = &matching[0];
int additions = 0, deletions = 0, substitutions = 0;

curr = &distance[curr1][curr2];

while (curr->p != 0) {
  switch (curr->p) {
  case 1:
    currmatch->c1 = '-';
    currmatch->mid = ' ';
    currmatch->c2 = seq2[length2-curr2];
    curr2--;
    deletions++;
```

```
        break;
      case 2:
        currmatch->c1 = seq1[length1-curr1];
        if (equal(curr1, curr2)) currmatch->mid = ':';
        else {
           currmatch->mid = ' ';
           substitutions++;
        }
        currmatch->c2 = seq2[length2-curr2];
        curr1--;
        curr2--;
        break;
      case 3:
        currmatch->c1 = seq1[length1-curr1];
        currmatch->mid = ' ';
        currmatch->c2 = '-';
        curr1--;
        additions++;
        break;
      }
      curr = &distance[curr1][curr2];
      matchlength++;
      currmatch++;
    }

    printf("\n");

    int currline, lengthleft = matchlength, offset = 0;

    while (lengthleft > 0) {
      currline = (lengthleft < 80) ? lengthleft : 80;
      lengthleft -= currline;

      /* write sequence 1 */
      currmatch = &matching[offset];
      for (i = 0; i < currline; i++) {
        printf("%c", currmatch->c1);
        currmatch++;
      }
      printf("\n");

      /* write middle */
      currmatch = &matching[offset];
      for (i = 0; i < currline; i++) {
        printf("%c", currmatch->mid);
        currmatch++;
    }
    printf("\n");

    /* write sequence 2 */
    currmatch = &matching[offset];
    for (i = 0; i < currline; i++) {
      printf("%c", currmatch->c2);
      currmatch++;
    }
    offset += currline;
    printf("\n\n");
  }

  if (additions > 0) printf("Additions:     %4d\n", additions);
  if (deletions > 0) printf("Deletions:     %4d\n", deletions);
  if (substitutions > 0) printf("Substitutions: %4d\n", substitutions);
  if (distance[length1][length2].d > 0)
    printf("\nTotal:         %4d\n\n", distance[length1][length2].d);
  else printf("Sequences identical.\n\n");

  fclose(seq1file);
  fclose(seq2file);

  exit(0);
}
```

```
/* Determines edit distance between two sequences of letters as specified
   in two argument files (ignores whitespace in sequences).  If one argument,
   reads sequence1 from standard input, sequence two from argument file.
   Uses a dynamic programming algorithm.  In order to make the final output
   simpler, runs sequences backwards along edges of matrix. */

extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
}

class entry {
 public:
  int d;      /* distance */
  int p;      /* previous - 0 for none, 1 for left, 2 for diagonal, 3 for up */
};

char *seq1, *seq2;
int length1, length2;

int equal(int i, int j) {
  return (seq1[length1-i] == seq2[length2-j]);
}

int min(int a, int b) {
  return (a < b? a : b);
}

main(int argc, char **argv) {
  FILE *seq1file, *seq2file;
  char c[2];

  entry **distance;  /* edit distance matrix */

  /* open sequence files */
  seq1file = fopen(argv[1], "r");
  seq2file = fopen(argv[2], "r");

  int buf1size = 512, buf2size = 512;

  /* read in sequences */
  seq1 = (char *) malloc(buf1size * sizeof(char));
  seq2 = (char *) malloc(buf2size * sizeof(char));

  /* read sequences in, ignoring whitespace */
  length1 = 0;
```

```
while (fscanf(seq1file, "%1s", c) == 1) {
  if (buf1size <= length1) {
    buf1size *= 2;
    seq1 = (char *) realloc(seq1, buf1size*sizeof(char));
  }
  seq1[length1++] = c[0];
}

length2 = 0;
while (fscanf(seq2file, "%1s", c) == 1) {
  if (buf2size <= length2) {
    buf2size *= 2;
    seq2 = (char *) realloc(seq2, buf2size*sizeof(char));
  }
  seq2[length2++] = c[0];
}

printf("\nBases: %4d\n       %4d\n", length1, length2);

/* allocate distance array */
distance = (entry **) calloc(length1+1, sizeof(entry *));

int i,j;

for (i=0; i <= length1; i++) {
  distance[i] = (entry *) calloc(length2+1, sizeof(entry));
}

entry *curr;

/* initialize (dummy) outer row and column */
curr = &distance[0][0];
curr->d = 0;
curr->p = 0;

for (j = 1; j <= length2; j++) {
  curr = &distance[0][j];
  curr->d = j;
  curr->p = 1;
}

for (i = 1; i <= length1; i++) {
  curr =.&distance[i][0];
  curr->d = i;
  curr->p = 3;
}
```

```
/* actual entries */
for (i = 1; i <= length1; i++) {
  for (j = 1; j <= length2; j++) {
    curr = &distance[i][j];
    /* first get minimum of upper and left neighbor */
    if (distance[i-1][j].d < distance[i][j-1].d) {
       curr->d = distance[i-1][j].d + 1;
       curr->p = 3;
    }
    else {
       curr->d = distance[i][j-1].d + 1;
       curr->p = 1;
    }

    /* if equal, see if diagonal entry is less, if not equal,
     see if diagonal entry + 1 is less */
    if ((distance[i-1][j-1].d + (equal(i,j)?0:1)) < curr->d) {
       curr->d = distance[i-1][j-1].d + (equal(i,j)?0:1);
       curr->p = 2;
    }
  }
}

/* starting at lower right corner of matrix, construct matching */
class match {
public:
  char c1;
  char mid;
  char c2;
};

match *matching;
matching = (struct match *) calloc(length1 + length2, sizeof(struct match));

int curr1 = length1, curr2 = length2, matchlength = 0;
match *currmatch = &matching[0];
int additions = 0, deletions = 0, substitutions = 0;

curr = &distance[curr1][curr2];

while (curr->p != 0) {
  switch (curr->p) {
  case 1:
    currmatch->c1 = '-';
    currmatch->mid = ' ';
    currmatch->c2 = seq2[length2-curr2];
    curr2--;
    deletions++;
    break;
  case 2:
    currmatch->c1 = seq1[length1-curr1];
    if (equal(curr1, curr2)) currmatch->mid = ':';
    else {
       currmatch->mid = ' ';
       substitutions++;
    }
    currmatch->c2 = seq2[length2-curr2];
    curr1--;
    curr2--;
    break;
  case 3:
    currmatch->c1 = seq1[length1-curr1];
    currmatch->mid = ' ';
    currmatch->c2 = '-';
    curr1--;
    additions++;
    break;
  }
  curr = &distance[curr1][curr2];
  matchlength++;
  currmatch++;
}

printf("\n");

int currline, lengthleft = matchlength, offset = 0;

while (lengthleft > 0) {
  currline = (lengthleft < 80) ? lengthleft : 80;
  lengthleft -= currline;

  /* write sequence 1 */
  currmatch = &matching[offset];
  for (i = 0; i < currline; i++) {
    printf("%c", currmatch->c1);
    currmatch++;
  }
  printf("\n");

  /* write middle */
  currmatch = &matching[offset];
```

```
      for (i = 0; i < currline; i++) {
        printf("%c", currmatch->mid);
        currmatch++;
      }
      printf("\n");

      /* write sequence 2 */
      currmatch = &matching[offset];
      for (i = 0; i < currline; i++) {
        printf("%c", currmatch->c2);
        currmatch++;
      }
      offset += currline;
      printf("\n\n");
    }

  if (additions > 0) printf("Additions:     %4d\n", additions);
  if (deletions > 0) printf("Deletions:     %4d\n", deletions);
  if (substitutions > 0) printf("Substitutions: %4d\n", substitutions);
  if (distance[length1][length2].d > 0)
    printf("\nTotal:         %4d\n\n", distance[length1][length2].d);
  else printf("Sequences identical.\n\n");

  fclose(seq1file);
  fclose(seq2file);

  exit(0);
}
```

```
extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
  #include <math.h>
}

#include "invert.h"

void copy (double a[4], double b[4]) {
  a[0] = b[0];
  a[1] = b[1];
  a[2] = b[2];
  a[3] = b[3];
}

int maxcol(double result[4]) {
  /* returns column number of max value */
  double max1, max2;

  max1 = (result[0] > result[1]) ? result[0] : result[1];
  max2 = (result[2] > result[3]) ? result[2] : result[3];
  if (max2 > max1) max1 = max2;

  if (max1 == result[0]) return 0;
  if (max1 == result[1]) return 1;
  if (max1 == result[2]) return 2;
  if (max1 == result[3]) return 3;

  /* default */
  return -1;
}

void findmax(FILE *datafile, int column, double *result) {
  /* find vectors with maximum value in column column (indexed by 0-3),
     subject to constraint that second max should be looked for excluding
     100 points surrounding first max. */

  double max[4];
  double curr[4];
  int maxtime, max2time, time = 0;

  /* initialize */
  for (time = 0; time < 200; time++)
    fscanf(datafile, "%*lf%*lf%*lf%*lf");

  fscanf(datafile, "%lf%lf%lf%lf", &curr[0], &curr[1], &curr[2], &curr[3]);
```

```
time++;
copy(max, curr);

fscanf(datafile, "%lf%lf%lf%lf", &curr[0], &curr[1], &curr[2], &curr[3]);
time++;
copy(max, curr);

while (fscanf(datafile, "%lf%lf%lf%lf",
                &curr[0], &curr[1], &curr[2], &curr[3]) != EOF) {
  time++;
  if ((curr[column] > max[column]) && (column == maxcol(curr))) {
    copy(max, curr);
    maxtime = time;
  }
}

fprintf(stderr, "    Time %4d:", maxtime);
fprintf(stderr, "%10.0lf %10.0lf %10.0lf %10.0lf\n",
         max[0], max[1], max[2], max[3]);
copy(result, max);

/* initialize */
time = 0;

fseek(datafile, 0L, SEEK_SET);

for (time = 0; time < 200; time++)
  fscanf(datafile, "%*lf%*lf%*lf%*lf");

fscanf(datafile, "%lf%lf%lf%lf", &curr[0], &curr[1], &curr[2], &curr[3]);
time++;
copy(max, curr);

while (fscanf(datafile, "%lf%lf%lf%lf",
                &curr[0], &curr[1], &curr[2], &curr[3]) != EOF) {
  time++;
  if ((abs(time - maxtime) > 50) && (curr[column] > max[column])
       && (column == maxcol(curr))) {
    copy(max, curr);
    max2time = time;
  }
}

fprintf(stderr, "    Time %4d:", max2time);
fprintf(stderr, "%10.0lf %10.0lf %10.0lf %10.0lf\n",
         max[0], max[1], max[2], max[3]);
  fseek(datafile, 0L, SEEK_SET);
  fprintf(stderr, "Yellow (A):\n");
  findmax(datafile, 2, max);
  for (i = 0; i < 4; i++)
    matrix[i][2] = max[i];

  fseek(datafile, 0L, SEEK_SET);
  fprintf(stderr, "Green  (G):\n");
  findmax(datafile, 3, max);
  for (i = 0; i < 4; i++)
    matrix[i][3] = max[i];

  fprintf(stderr, "Matrix:\n");
  for (i = 0 ; i < 4; i++) {
    for (j = 0; j < 4; j++)
      fprintf(stderr, "%10.0lf ", matrix[i][j]);
    fprintf(stderr, "\n");
  }

  invert(matrix, inversion, 4);

  fprintf(stderr, "Inverting matrix ... \n");
  for (i = 0 ; i < 4; i++) {
    for (j = 0; j < 4; j++)
      printf("%15.10lf ", inversion[i][j]);
    printf("\n");
  }
  fprintf(stderr, "Done.\n");
}
```

```
  result[0] = (result[0] + max[0])/2;
  result[1] = (result[1] + max[1])/2;
  result[2] = (result[2] + max[2])/2;
  result[3] = (result[3] + max[3])/2;
}


main(int argc, char *argv[]) {
  if (argc < 2) {
    fprintf(stderr, "Usage: %s <datafile> \n", argv[0]);
    exit(1);
  }

  malloc_debug(8);

  FILE *datafile;

  /* read from argument file */
  if (!(datafile = fopen(argv[1], "r"))) {
    fprintf(stderr, "%s: can not open file %s\n", argv[0], argv[1]);
    exit(1);
  }

  double max[4];
  double **matrix, **inversion;
  int i,j;

  matrix = (double **) calloc(4, sizeof(double *));
  inversion = (double **) calloc(4, sizeof(double *));

  for (i = 0; i < 4; i++) {
    matrix[i] = (double *) calloc(4, sizeof(double));
    inversion[i] = (double *) calloc(4, sizeof(double));
  }

  fprintf(stderr, "Purple (C):\n");
  findmax(datafile, 0, max);
  for (i = 0; i < 4; i++)
    matrix[i][0] = max[i];

  fseek(datafile, 0L, SEEK_SET);
  fprintf(stderr, "Red    (T):\n");
  findmax(datafile, 1, max);
  for (i = 0; i < 4; i++)
    matrix[i][1] = max[i];
```

```
/* Reads tags off a four channel tagged file and writes them to
   standard output.*/

extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
}

main(int argc, char **argv) {

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <tagged file>\n", argv[0]);
    exit(1);
  }

  FILE *taggedfile;

  taggedfile = fopen(argv[1], "r");

  if (!taggedfile) {
    fprintf(stderr, "%s: Can not open tagged file %s.\n", argv[0], argv[1]);
    exit(1);
  }

  double c, a, g, t;
  char call[2];
  int i = 0;

  while ((fscanf(taggedfile, "%*lf %*lf %*lf %*lf %1s", call) == 1)) {
    if (call[0] != ‘X’) {
      i++;
      printf("%c", call[0]);
      if (!(i%10)) printf(" ");
      if (!(i%40)) printf("\n");
    }
  }

  printf("\n");

  fclose(taggedfile);
  exit(0);
}
```

```
  dist =(
//          pow((vec->threeprev.c - average.threeprev.c), 2) +
//          pow((vec->threeprev.a - average.threeprev.a), 2) +
//          pow((vec->threeprev.g - average.threeprev.g), 2) +
//          pow((vec->threeprev.t - average.threeprev.t), 2) +
            pow((vec->prev.c - average.prev.c)*1.5, 2) +
            pow((vec->prev.a - average.prev.a)*1.5, 2) +
            pow((vec->prev.g - average.prev.g)*1.5, 2) +
            pow((vec->prev.t - average.prev.t)*1.5, 2) +
            pow((vec->curr.c - average.curr.c)*2, 2) +
            pow((vec->curr.a - average.curr.a)*2, 2) +
            pow((vec->curr.g - average.curr.g)*2, 2) +
            pow((vec->curr.t - average.curr.t)*2, 2) +
            pow((vec->next.c - average.next.c)*1.5, 2) +
            pow((vec->next.a - average.next.a)*1.5, 2) +
            pow((vec->next.g - average.next.g)*1.5, 2) +
            pow((vec->next.t - average.next.t)*1.5, 2) +
            pow(vec->prev.c - average.prev.c + vec->next.c - average.next.c, 2) +
            pow(vec->prev.a - average.prev.a + vec->next.t - average.next.t, 2) +
            pow(vec->prev.g - average.prev.g + vec->next.a - average.next.a, 2) +
            pow(vec->prev.t - average.prev.t + vec->next.g - average.next.g, 2) +
//          pow((vec->threenext.c - average.threenext.c), 2) +
//          pow((vec->threenext.a - average.threenext.a), 2) +
//          pow((vec->threenext.g - average.threenext.g), 2) +
//          pow((vec->threenext.t - average.threenext.t), 2) +
            pow(vec->threeprev.c - average.threeprev.c +
                vec->threenext.c - average.threenext.c, 2) +
            pow(vec->threeprev.a - average.threeprev.a +
                vec->threenext.t - average.threenext.t, 2) +
            pow(vec->threeprev.g - average.threeprev.g +
                vec->threenext.a - average.threenext.a, 2) +
            pow(vec->threeprev.t - average.threeprev.t +
                vec->threenext.g - average.threenext.g, 2) +
            pow((vec->timetocall - average.timetocall)*.3, 2) +
            pow((vlast - alast), 2) +
            pow((log(vmax) - log(amax))*.5, 2) +
            pow((log(vec->threeprevmax) - log(average.threeprevmax))*.25, 2) +
            pow((log(vec->threenextmax) - log(average.threenextmax))*.25, 2)
            );


  dist =((
//           pow((vec->threeprev.c - average.threeprev.c), 2) +
//           pow((vec->threeprev.a - average.threeprev.a), 2) +
//           pow((vec->threeprev.g - average.threeprev.g), 2) +
//         pow((vec->threeprev.t - average.threeprev.t), 2) +
           pow((vec->prev.c - average.prev.c)*1.5, 2) +
           pow((vec->prev.a - average.prev.a)*1.5, 2) +
           pow((vec->prev.g - average.prev.g)*1.5, 2) +
           pow((vec->prev.t - average.prev.t)*1.5, 2) +
           pow((vec->curr.c - average.curr.c)*2.0, 2) +
           pow((vec->curr.a - average.curr.a)*2.0, 2) +
           pow((vec->curr.g - average.curr.g)*2.0, 2) +
           pow((vec->curr.t - average.curr.t)*2.0, 2) +
           pow((vec->next.c - average.next.c)*1.5, 2) +
           pow((vec->next.a - average.next.a)*1.5, 2) +
           pow((vec->next.g - average.next.g)*1.5, 2) +
           pow((vec->next.t - average.next.t)*1.5, 2) +
           pow(vec->prev.c - average.prev.c + vec->next.c - average.next.c, 2) +
           pow(vec->prev.a - average.prev.a + vec->next.t - average.next.t, 2) +
           pow(vec->prev.g - average.prev.g + vec->next.a - average.next.a, 2) +
           pow(vec->prev.t - average.prev.t + vec->next.g - average.next.g, 2) +
//         pow((vec->threenext.c - average.threenext.c), 2) +
//         pow((vec->threenext.a - average.threenext.a), 2) +
//         pow((vec->threenext.g - average.threenext.g), 2) +
//         pow((vec->threenext.t - average.threenext.t), 2) +
           pow(vec->threeprev.c - average.threeprev.c +
               vec->threenext.c - average.threenext.c, 2) +
           pow(vec->threeprev.a - average.threeprev.a +
               vec->threenext.t - average.threenext.t, 2) +
           pow(vec->threeprev.g - average.threeprev.g +
               vec->threenext.a - average.threenext.a, 2) +
           pow(vec->threeprev.t - average.threeprev.t +
               vec->threenext.g - average.threenext.g, 2)) *
          (pow(0.3*(vec->timetocall - average.timetocall), 2) + 1) //*
//        (pow(0.01*(vmax/vlast - amax/alast), 2) + 1)
//        ((vmax < amax)? (pow(0.3*(vmax - amax), 2) + 1) : 1)// +
//        pow((log(vec->threeprevmax) - log(average.threeprevmax))*.25, 2) +
//        pow((log(vec->threenextmax) - log(average.threenextmax))*.25, 2)
          );
```

```
/* Routines for matrix inversion, from Numerical Recipes in C, p.43-45,
   but with indexing running from 0 to n-1 */

extern "C" {
  #include <math.h>
  #include <stdio.h>
  #include <stdlib.h>
}

#define TINY 1.0e-20;

void nrerror(char error_text[]) {
/* Numerical Recipes standard error handler */

  fprintf(stderr, "Numerical Recipes run-time error...\n");
  fprintf(stderr, "%s\n", error_text);
  exit(1);
};

double *vector(int nl, int nh) {
  double *v;

  v = (double *) malloc((unsigned) (nh-nl+1) * sizeof(double));
  if (!v) nrerror("allocation failure in vector()");
  return v-nl;
}

void free_vector(double *v, int nl, int nh) {
  free((char *) (v+nl));
}

void ludcmp(double **a, int n, int *indx, double *d) {
  int i, imax, j, k;
  double big, dum, sum, temp;
  double *vv;

  vv = vector(0,n-1);
  *d = 1.0;
  for (i=0; i<n; i++) {
    big=0.0;
    for (j=0; j<n; j++)
      if ((temp=fabs(a[i][j])) > big) big=temp;
    if (big == 0.0) nrerror("Singular matrix in routine LUDCMP");
    vv[i] = 1.0/big;
  }
  for (j=0;j<n;j++) {
    for (i=0;i<j;i++) {
      sum = a[i][j];
      for (k=0;k<i;k++) sum -= a[i][k]*a[k][j];
      a[i][j] = sum;
    }
    big = 0.0;
    for (i=j;i<n;i++) {
      sum = a[i][j];
      for (k=0;k<j;k++)
        sum -= a[i][k]*a[k][j];
      a[i][j] = sum;
      if ((dum = vv[i]*fabs(sum)) >= big) {
        big=dum;
        imax=i;
      }
    }
    if (j != imax) {
      for (k=0; k<n; k++) {
        dum = a[imax][k];
        a[imax][k] = a[j][k];
        a[j][k] = dum;
      }
      *d = -(*d);

      vv[imax]=vv[j];
    }
    indx[j]=imax;
    if (a[j][j] == 0.0) a[j][j] = TINY;
    if (j != n) {
      dum = 1.0 / (a[j][j]);
      for (i = j+1; i<n; i++) a[i][j] *= dum;
    }
  }
  free_vector(vv,0,n-1);
}

void lubksb(double **a, int n, int *indx, double b[]) {
  int i, ii=-1, ip, j;
  double sum;

  for (i=0; i<n; i++) {
    ip=indx[i];
    sum=b[ip];
    b[ip]=b[i];
    if (ii)
      for (j=ii; j<=i-1; j++) sum -= a[i][j]*b[j];
    else if (sum) ii=i;
    b[i]=sum;
  }
  for (i=n-1; i>=0; i--) {
    sum = b[i];
    for (j=i+1; j < n; j++) sum -= a[i][j]*b[j];
    b[i]=sum/a[i][i];
  }
}

void invert(double **a, double **y, int n) {
  int *indx, i, j;
  double *col, d;

  indx = (int *) calloc(n, sizeof(int));
  col = (double *) calloc(n, sizeof(double));

  ludcmp(a, n, indx, &d);

  for (j=0; j < n; j++) {
    for (i=0; i < n; i++) col[i]=0.0;
    col[j] = 1.0;
    lubksb(a, n, indx, col);
    for (i=0;i<n;i++) y[i][j] = col[i];
  }
}
```

```
/* Fits a line to a set of points.  Taken from  Numerical Recipes in C,
 *  p. 527.
 */

extern "C" {
  #include <math.h>
  #include <stdio.h>
  #include <stdlib.h>
}

static double sqrarg;
#define SQR(a) (sqrarg=(a), sqrarg*sqrarg)
#define ITMAX 100
#define EPS 3.0e-7

void nrerror(char error_text[]) {
/* Numerical Recipes standard error handler */

  fprintf(stderr, "Numerical Recipes run-time error...\n");
  fprintf(stderr, "%s\n", error_text);
  exit(1);
};

double gammln(double xx) {
/* Returns the value ln(floor(gamma(xx))) for xx > 0.  Full accuracy is
   obtained for cc > 1.  For 0 < xx < 1, the reflection formula can be
   used first. */

  double x, tmp, ser;
  static double cof[6] = {76.18009173, -86.50532033, 24.01409822,
                          -1.231739516, 0.120858003e-2, -0.536382e-5};
  int j;

  x = xx-1.0;
  tmp = x + 5.5;
  tmp -= (x + 0.5) * log(tmp);
  ser = 1.0;
  for (j=0; j<=5; j++) {
    x += 1.0;
    ser += cof[j]/x;
  }
  return -tmp + log(2.50662827465 * ser);
};

void gser(double *gamser, double a, double x, double *gln) {
/* Returns the incomplete gamma function P(a, x) evaluated by its series
```

```
   representation as gamser.  Also returns ln(gamma(a)) as gln. */

  int n;
  double sum, del, ap;

  *gln = gammln(a);
  if (x <= 0.0) {
    if (x < 0.0) nrerror("x less than 0 in GSER");
    *gamser=0.0;
    return;
  }
  else {
    ap = a;
    del = sum = 1.0/a;
    for (n = 1; n <= ITMAX; n++) {
      ap += 1.0;
      del *= x/ap;
      sum += del;
      if (fabs(del) < fabs(sum)*EPS) {
         *gamser = sum * exp(-x + a * log(x) - (*gln));
         return;
      }
    }
    nrerror("a too large, ITMAX too small in GSER");
  }
};

void gcf(double *gammcf, double a, double x, double *gln) {
/* Returns the incomplete gamma function Q(a, x) evaluated by its continued
   fraction representation as gammcf.  Also returns ln(gamma(a)) as gln. */

  int n;
  double gold = 0.0, g, fac = 1.0, b1 = 1.0;
  double b0 = 0.0, anf, ana, an, a1, a0 = 1.0;

  *gln = gammln(a);
  a1 = x;
  for (n = 1; n <= ITMAX; n++) {
    an = (double) n;
    ana = an - a;
    a0 = (a1 + a0 * ana) * fac;
    b0 = (b1 + b0 * ana) * fac;
    anf = an * fac;
    a1 = x * a0 + anf * a1;
    b1 = x * b0 + anf * b1;
    if (a1) {
      fac = 1.0/a1;
      g = b1 * fac;
      if (fabs((g-gold)/g) < EPS) {
        *gammcf = exp(-x + a * log(x) - (*gln))*g;
        return;
      }
      gold = g;
    }
  }
  nrerror("a too large, ITMAX too small in routine GCF");
};

double gammq(double a, double x) {
/* Returns the incomplete gamma function Q(a, x) = 1 - P(a, x) */
  double gamser, gammcf, gln;

  if (x < 0.0 || a <= 0.0) nrerror("Invalid arguments in GAMMQ");
  if (x < (a + 1.0)) {     /* use the series representation */
    gser(&gamser, a, x, &gln);
    return 1.0 - gamser;   /* and take its complement */
  }
  else {                   /* use the continued fraction representation */
    gcf(&gammcf, a, x, &gln);
    return gammcf;
  }
};

void fit(double x[], double y[], int ndata, double sig[], int mwt,
          double *a, double *b, double *siga, double *sigb,
          double *chi2, double *q) {
/* Given a set of points x[1..ndata], y[1..ndata] with standard deviations
    sig[1..ndata], fit them to a straight line y=ax+b by minimizing
    chi-squared.  Returned are a, b, and their respective probable
    uncertainties siga and sigb, the chi-square chi2, and the goodness-of-fit
    probability q (that the fit would have chi-square this large or larger).
    If mwt=0 on input, then the standard deviations are assumed to be
    unavailable: q is returned as 1.0 and the normalization of chi2 is to unit
    standard deviation on all points. */

   int i;
   double wt, t, sxoss, sx = 0.0, sy = 0.0, st2 = 0.0, ss, sigdat;

   *b = 0.0;
   if (mwt) {
     ss = 0.0;
     for (i=1; i <= ndata; i++) {
      wt = 1.0/SQR(sig[i]);
      ss += wt;
      sx += x[i]*wt;
      sy += y[i]*wt;
    }
  }
  else {
    for (i=1; i <= ndata; i++) {
      sx += x[i];
      sy += y[i];
    }
    ss = ndata;
  }
  sxoss = sx/ss;
  if (mwt) {
    for (i=1; i < ndata; i++) {
      t = (x[i] - sxoss)/sig[i];
      st2 += t*t;
      *b += t * y[i] / sig[i];
    }
  }
  else {
    for (i=1; i <= ndata; i++) {
      t = x[i] - sxoss;
      st2 += t*t;
      *b += t*y[i];
    }
  }
  *b /= st2;
  *a = (sy-sx*(*b))/ss;
  *siga = sqrt((1.0+sx*sx/(ss*st2))/ss);
  *sigb = sqrt(1.0/st2);
  *chi2 = 0.0;
  if (mwt == 0) {
    for (i=1; i <= ndata; i++)
      *chi2 += SQR(y[i]-(*a)-(*b)*x[i]);
    *q = 1.0;
    sigdat = sqrt((*chi2)/(ndata-2));
    *siga *= sigdat;
    *sigb *= sigdat;
  }
  else {
    for (i=1; i <=ndata; i++)
      *chi2 += SQR((y[i]-(*a)-(*b)*x[i])/sig[i]);
    *q = gammq(0.5 * (ndata - 2), 0.5*(*chi2));
  }
```

```
/* Multiplies four channel file by four x four matrix, one vector at a time.
   If two arguments, considers them data file and matrix file.  If one,
   considers it matrix file, and reads data from standard input.  Writes
   to standard output. */

extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
}

void matrix_vector_multiply(double **matrix, double *vector,
                               int n, double *result) {
  /* multiply nxn matrix by length n vector and put result in
     (already allocated) length n vector result */
  int i, j;
  double sum;

  for (i = 0; i < n; i++) {
    sum = 0;
    for (j = 0; j < n; j++) {
      sum += (matrix[i][j] * vector[j]);
    }
    result[i] = sum;
  }
}

void vector_matrix_multiply(double **matrix, double *vector,
                               int n, double *result) {
  /* multiply length n vector by nxn matrix by and put result in
     (already allocated) length n vector result */
  int i, j;
  double sum;

    for (j = 0; j < n; j++) {
    sum = 0;
    for (i = 0; i < n; i++) {
      sum += (matrix[i][j] * vector[i]);
    }
    result[j] = sum;
  }
}

main(int argc, char **argv) {
  if (argc < 2) {
    fprintf(stderr, "Usage: %s <matrix file> <datafile> \n", argv[0]);
    exit(1);
  }

  FILE *datafile, *matrixfile;

  matrixfile = fopen(argv[1], "r");

  if (!matrixfile) {
    fprintf(stderr, "%s: Can not open matrix file %s.\n", argv[0], argv[1]);
    exit(1);
  }

  if (argc == 3) {
    datafile = fopen(argv[2], "r");

    if (!datafile) {
      fprintf(stderr, "%s: Can not open data file %s.\n", argv[0], argv[2]);
      exit(1);
    }
  }
  else datafile = stdin;

  double **matrix, *currvec, *result;
  int i, j;

  matrix = (double **) calloc(4, sizeof(double *));
  currvec = (double *) calloc(4, sizeof(double));
  result = (double *) calloc(4, sizeof(double));

  for (i=0; i<4; i++)
    matrix[i] = (double *) calloc(4, sizeof(double));

  for (i=0; i<4; i++)
    for (j=0; j<4; j++) {
      if (fscanf(matrixfile, "%lf", &matrix[i][j]) != 1) {
         fprintf(stderr, "%s: matrix file %s incomplete \n", argv[0], argv[1]);
         exit(1);
      }
    }

  fclose(matrixfile);

/*
  printf("Matrix: \n");
  for (i=0; i<4; i++) {
    for (j=0; j<4; j++) {
      printf("%10f", matrix[i][j]);
```

```
}

FILE *datafile, *matrixfile;

matrixfile = fopen(argv[1], "r");

if (!matrixfile) {
  fprintf(stderr, "%s: Can not open matrix file %s.\n", argv[0], argv[1]);
  exit(1);
}

if (argc == 3) {
  datafile = fopen(argv[2], "r");

  if (!datafile) {
    fprintf(stderr, "%s: Can not open data file %s.\n", argv[0], argv[2]);
    exit(1);
  }
}
else datafile = stdin;

double **matrix, *currvec, *result;
int i, j;

matrix = (double **) calloc(4, sizeof(double *));
currvec = (double *) calloc(4, sizeof(double));
result = (double *) calloc(4, sizeof(double));

for (i=0; i<4; i++)
  matrix[i] = (double *) calloc(4, sizeof(double));

for (i=0; i<4; i++)
  for (j=0; j<4; j++) {
    if (fscanf(matrixfile, "%lf", &matrix[i][j]) != 1) {
      fprintf(stderr, "%s: matrix file %s incomplete \n", argv[0], argv[1]);
      exit(1);
    }
  }

fclose(matrixfile);

/*
 printf("Matrix: \n");
 f r (i=0; i<4; i++) {
   for (j=0; j<4; j++) {
     printf("%10f", matrix[i][j]);
    }
    printf("\n");
  }
  printf("\n");
*/

  while (fscanf(datafile, "%lf%lf%lf%lf",
                &currvec[0], &currvec[1], &currvec[2], &currvec[3]) == 4) {
    matrix_vector_multiply(matrix, currvec, 4, result);
    printf("%10.6f %10.6f %10.6f %10.6f\n",
            result[0], result[1], result[2], result[3]);
  }

  fclose(datafile);

  exit(0);
}
```

```
/* Contains member functions for vector class, as well a
   other auxilliary functions. */

#include "prototype.h"

extern double currmin;

void datapoint::print(FILE *fp, int flag) {
  /* if flag is one, print tag; if zero don't print tag */
  fprintf(fp, "%8.0f", c);
  fprintf(fp, "%8.0f", t);
  fprintf(fp, "%8.0f", a);
  fprintf(fp, "%8.0f", g);
  if (flag) fprintf(fp, "%8c", inttocall(tag));
  fprintf(fp, "\n");
}

void datapoint::init(double cp, double tp, double ap, double gp, int tg) {
  c = cp;
  t = tp;
  a = ap;
  g = gp;
  tag = tg;
}

void vector::print(FILE *fp) {
  prev.print(fp);
  curr.print(fp);
  next.print(fp);
  fprintf(fp, "%3c%10.0f\n", inttocall(tag), max);
}

void vector::normalize() {
  /* normalize all fluorescence values so that max value is 100, and set
     max field to absolute max.  */

  double max1, max2, max3, max4, max5, max6;  /* quick hack! */

  /* round one - compare pairs of values */
  max1 = (prev.c > prev.a)? prev.c : prev.a;
  max2 = (prev.g > prev.t)? prev.g : prev.t;
  max3 = (curr.c > curr.a)? curr.c : curr.a;
  max4 = (curr.g > curr.t)? curr.g : curr.t;
  max5 = (next.c > next.a)? next.c : next.a;
  max6 = (next.g > next.t)? next.g : next.t;
  /* round two - compare winners of round 1 */
  max1 = (max1 > max2)? max1 : max2;
  max2 = (max3 > max4)? max3 : max4;
  max3 = (max5 > max6)? max5 : max6;

  /* round three - determine largest winner of round 2 */
  max1 = (max1 > max2)? max1 : max2;
  max1 = (max1 > max3)? max1 : max3;
  max = max1;

  /* max1 is now max.  Normalize values */
  if (max1 != 0) {
    prev.c = (prev.c * 100.0) / max1;
    prev.a = (prev.a * 100.0) / max1;
    prev.g = (prev.g * 100.0) / max1;
    prev.t = (prev.t * 100.0) / max1;
    curr.c = (curr.c * 100.0) / max1;
    curr.a = (curr.a * 100.0) / max1;
    curr.g = (curr.g * 100.0) / max1;
    curr.t = (curr.t * 100.0) / max1;
    next.c = (next.c * 100.0) / max1;
    next.a = (next.a * 100.0) / max1;
    next.g = (next.g * 100.0) / max1;
    next.t = (next.t * 100.0) / max1;
  }
}

int iscall(int call) {
  /* determines whether a data line has been called */
  return((call >= 0) && (call <= 4));
}

int calltoint(char call) {
  /* converts character tag to int */
  switch (call) {
  case 'C':
    return 0;
  case 'A':
    return 1;
  case 'G':
    return 2;
  case 'T':
    return 3;
  case 'X':
  default:
    return -1;
  }
}

char inttocall(int tag) {
  switch (tag) {
  case 0:
    return ‘C’;
  case 1:
    return ‘A’;
  case 2:
    return ‘G’;
  case 3:
    return ‘T’;
  case -1:
  default:
    return ‘.’;
  }
}

double vector::distance(vector *vec) {
  double dist, vmax, amax;

  if (vec->max <= 0) vmax = 1;
  else vmax = vec->max;
  if (this->max <= 0) amax = 1;
  else amax = this->max;

  dist =(pow(vec->prev.c - this->prev.c, 2) +
          pow(vec->prev.a - this->prev.a, 2) +
          pow(vec->prev.g - this->prev.g, 2) +
          pow(vec->prev.t - this->prev.t, 2) +
          pow(vec->curr.c - this->curr.c, 2) +
          pow(vec->curr.a - this->curr.a, 2) +
          pow(vec->curr.g - this->curr.g, 2) +
          pow(vec->curr.t - this->curr.t, 2) +
          pow(vec->next.c - this->next.c, 2) +
          pow(vec->next.a - this->next.a, 2) +
          pow(vec->next.g - this->next.g, 2) +
          pow(vec->next.t - this->next.t, 2) +
//        pow((vec->timetocall - this->timetocall)*20, 2) +
          pow((log(vmax) - log(amax))*50, 2)
          );

  return(dist);
}
```

```
int vector::call(vector *prot) {
  /* determines tag of closest prototype */

  double dist[NUMPROT];
  int i;

  for (i=0; i < NUMPROT; i++) {
    dist[i] = prot[i].distance(this);
  }

  /* note: for coding purposes, use linear min.  Should change
     to log time min by pairing */
  double min;

  /* go through array updating min as you go */
  min = dist[0];
  for (i = 1; i < NUMPROT; i++) {
    if (dist[i] < min) min = dist[i];
  }

  /* min is now minimum distance - determine call */
  /* set currmin to this value */
  currmin = min;
  //  fprintf(stderr, "\n%15.0f ", min);
  for (i = 0; i < NUMPROT; i++) {
    if (min == dist[i]) {
/*      fprintf(stderr, "Current vector: \n");
      this->print();
      fprintf(stderr, "Closest prototype: \n");
      prot[i].print();
      fprintf(stderr, "(%d) ", i); */
      return (i);
    }
  }

  /* default */
  return(-1);
}

int avetimetocall(int prev, int curr) {
  switch (4*prev + curr) {
  case 0:
  case 2:
  case 6:
    return 6;
  case 1:
```

```
  case 3:
  case 4:
  case 5:
  case 7:
  case 10:
  case 12:
  case 14:
  case 15:
    return 7;
  case 8:
  case 11:
  case 13:
    return 8;
  case 9:
    return 9;
  }
  return 0;
}

void vector::movetowards(vector *vec, double movefactor) {
  /* move this towards vec by factor movefactor */
  prev.c += (vec->prev.c - prev.c)*movefactor;
  prev.t += (vec->prev.t - prev.t)*movefactor;
  prev.a += (vec->prev.a - prev.a)*movefactor;
  prev.g += (vec->prev.g - prev.g)*movefactor;
  curr.c += (vec->curr.c - curr.c)*movefactor;
  curr.t += (vec->curr.t - curr.t)*movefactor;
  curr.a += (vec->curr.a - curr.a)*movefactor;
  curr.g += (vec->curr.g - curr.g)*movefactor;
  next.c += (vec->next.c - next.c)*movefactor;
  next.t += (vec->next.t - next.t)*movefactor;
  next.a += (vec->next.a - next.a)*movefactor;
  next.g += (vec->next.g - next.g)*movefactor;
  max += (vec->max - max)*movefactor;

  return;
}

void vector::moveaway(vector *vec, double movefactor) {
/* move this away from vec by movefactor */
  prev.c -= (vec->prev.c - prev.c)*movefactor;
  prev.t -= (vec->prev.t - prev.t)*movefactor;
  prev.a -= (vec->prev.a - prev.a)*movefactor;
  prev.g -= (vec->prev.g - prev.g)*movefactor;
  curr.c -= (vec->curr.c - curr.c)*movefactor;
  curr.t -= (vec->curr.t - curr.t)*movefactor;
  curr.a -= (vec->curr.a - curr.a)*movefactor;
  curr.g -= (vec->curr.g - curr.g)*movefactor;
  next.c -= (vec->next.c - next.c)*movefactor;
  next.t -= (vec->next.t - next.t)*movefactor;
  next.a -= (vec->next.a - next.a)*movefactor;
  next.g -= (vec->next.g - next.g)*movefactor;
  max -= (vec->max - max)*movefactor;

  return;
}
```

```
/* Reads tags off a four channel tagged file and copies them on to a
   four channel untagged file.  First argument is tagged file.  Second, if
   there is one, is untagged file.  Otherwise, reads untagged file
   from standard input.  Writes to standard output.*/

extern "C" {
  #include <stdio.h>
  #include <stdlib.h>


main(int argc, char **argv) {
  if (argc < 2) {
    fprintf(stderr, "Usage: %s <tagged file> <untagged file> \n", argv[0]);
    exit(1);
  }

  FILE *taggedfile, *untaggedfile;

  taggedfile = fopen(argv[1], "r");

  if (!taggedfile) {
    fprintf(stderr, "%s: Can not open tagged file %s.\n", argv[0], argv[1]);
    exit(1);
  }

  if (argc == 3) {
    untaggedfile = fopen(argv[2], "r");

    if (!untaggedfile) {
      fprintf(stderr, "%s: Can not open untagged file %s.\n",
                argv[0], argv[2]);
      exit(1);
    }
  }
  else untaggedfile = stdin;

  double c, a, g, t;
  char call[2];

  while ((fscanf(taggedfile, "%*lf %*lf %*lf %*lf %1s", call) == 1) &&
          fscanf(untaggedfile, "%lf %lf %lf %lf", &c, &t, &a, &g)) {
    printf("%10.6f %10.6f %10.6f %10.6f     %c\n",
             c, t, a, g, call[0]);
  }

  fclose(taggedfile);
  fclose(untaggedfile);

  exit(0);
}
```

```
#include "cluster.h"

int main(int argc, char *argv[]) {
  int time;           /* current time */
  int basenum=0;
  int timeofcall;     /* time of last base call */
  int lastcall;       /* last base call */
  int i;
  char ctag[2];       /* for reading tag character */
  double cmax, tmax, amax, gmax;

  FILE *datafile;

  if (argc == 1) datafile = stdin;
  else {
    if (!(datafile = fopen(argv[1], "r"))) {
      fprintf(stderr, "Can not open data file %s.\n", argv[1]);
      exit(1);
    }
  }

  datapoint prev, curr, next; /* previous, current, and next data points */
  double threshold = 0;
  int timesincetag = 0;
  double multiplier;

  /* initialize prev and curr to first two time points */
  multiplier = .0204*(14*timesincetag - timesincetag*timesincetag);
  fscanf(datafile, "%lf%lf%lf%lf%ls",
          &prev.c, &prev.t, &prev.a, &prev.g, ctag);
  prev.tag = calltoint(ctag[0]);
  prev.c *= multiplier;
  prev.t *= multiplier;
  prev.a *= multiplier;
  prev.g *= multiplier;
  fscanf(datafile, "%lf%lf%lf%lf%ls",
          &curr.c, &curr.t, &curr.a, &curr.g, ctag);
  curr.tag = calltoint(ctag[0]);
  timesincetag++;
  multiplier = .0204*(14*timesincetag - timesincetag*timesincetag);
  curr.c *= multiplier;
  curr.t *= multiplier;
  curr.a *= multiplier;
  curr.g *= multiplier;
  time = 2;
  lastcall = 3;
  timeofcall = 0;

  fprintf(stderr, "Reading file...\n");
  while ((fscanf(datafile, "%lf%lf%lf%lf%ls",
                   &next.c, &next.t, &next.a, &next.g, ctag))
          != EOF) {
    next.tag = calltoint(ctag[0]);
    if (basenum == 0)    threshold = 0.145;
//    if (basenum == 150) threshold = 0.065;
//    if (basenum == 300) threshold = 0.06;
//    if (basenum == 450) threshold = 0.055;
//    if (basenum == 600) threshold = 0.05;

    timesincetag++;
    multiplier = .0204*(14*timesincetag - timesincetag*timesincetag);
    next.c *= multiplier;
    next.t *= multiplier;
    next.a *= multiplier;
    next.g *= multiplier;

    /* call according to channel that is highest of those channels that
       have local maxima at current time (no call if none) */
    cmax = ((curr.c >= prev.c) && (curr.c > next.c) && (curr.c > threshold))?
      curr.c : 0;
    tmax = ((curr.t >= prev.t) && (curr.t > next.t) && (curr.t > threshold))?
      curr.t : 0;
    amax = ((curr.a >= prev.a) && (curr.a > next.a) && (curr.a > threshold))?
      curr.a : 0;
    gmax = ((curr.g >= prev.g) && (curr.g > next.g) && (curr.g > threshold))?
      curr.g : 0;

    if ((cmax > 0) || (tmax > 0) || (amax > 0) || (gmax > 0)) {
      if ((basenum % 10) == 0) printf(" ");
      if ((basenum % 40) == 0) printf("\n");
      basenum++;
      timesincetag = 0;
      /* find largest max */
      if (cmax > tmax) {
        if (cmax > amax) {
          if (cmax > gmax) {
            printf("C");
          } else {
            printf("G");
          }
        } else {
          if (amax > gmax) {
            printf("A");
          } else {
            printf("G");
          }
        }
      }
      else {
        if (tmax > amax) {
          if (tmax > gmax) {
            printf("T");
          } else {
            printf("G");
          }
        } else {
          if (amax > gmax) {
            printf("A");
          } else {
            printf("G");
          }
        }
      }
    }

    /* move to next time */
    time++;
    prev = curr;
    curr = next;
  }

  printf("\n");

  fclose(datafile);
  exit(0);
}
```

```
#include "cluster.h"

#define TAGFILE 0
   /* if nonzero, writes a new tagged file with obtained tags */
#define TAGGEDFILE 0
   /* if 1, expects input file to be tagged with character tags */
#define FULLOUTPUT 0
   /* if nonzero, prints out distances and closest cluster */

/* Call data by closest cluster. */

extern double currmin;
extern int currcluster;

int main(int argc, char *argv[]) {
  int time=0;         /* current time */
  int basenum=0;      /* number of bases called */
  int timeofcall;     /* time of last base call */
  int lastcall;       /* last base call */
  double lastcallval; /* value of call channel at last call */
  int i;

  FILE *datafile, *clusterfile;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <cluster file> <data file>\n", argv[0]);
    exit(1);
  }

  if (!(clusterfile = fopen(argv[1], "r"))) {
    fprintf(stderr, "Can not open cluster file %s.\n", argv[1]);
    exit(1);
  }

  if (argc < 3) datafile = stdin;
  else {
    if (!(datafile = fopen(argv[2], "r"))) {
      fprintf(stderr, "Can not open data file %s.\n", argv[2]);
      exit(1);
    }
  }

  /* read in clusters - later, fix to have only cluster averages and
     check for premature end of cluster file */
  cluster *clust[NUMCLUSTS];  /* clusters */
  cluster *rv;
```

```
/* read through, collecting calls */
for (i = 0; i < NUMCLUSTS; i++) {  /* initialize */
  if ((rv = (cluster *) malloc(sizeof(cluster))) != NULL) {
    clust[i] = rv;
    clust[i]->average.input(clusterfile);
  }
  else {
    fprintf(stderr, "\nNot enough memory - Cluster %d\n", i);
    exit(1);
  }
}

datapoint threeprev, twoprev, twonext, threenext;
datapoint prev, curr, next; /* previous, current, and next data points */
vector vec;                 /* current vector */


char call;
double twoprevmin = 0, prevmin = 0;
int prevcluster;
int prevcall = -1;
double prevval = 1;
int disttocall = 0;

/* initialize threeprev through twonext to first six time points */
threeprev.input(datafile, TAGGEDFILE);
twoprev.input(datafile, TAGGEDFILE);
prev.input(datafile, TAGGEDFILE);
curr.input(datafile, TAGGEDFILE);
next.input(datafile, TAGGEDFILE);
twonext.input(datafile, TAGGEDFILE);
time = 2;
lastcall = 0;
lastcallval = 1;
timeofcall = 0;

if (TAGFILE) {
  threeprev.tag = -1;
  twoprev.tag = -1;
  prev.tag = -1;
  curr.tag = -1;
  next.tag = -1;
  twonext.tag = -1;

  threeprev.print(stdout, 1);
```

```
    twoprev.print(stdout, 1);
  }

  /* read through */
  fprintf(stderr, "Calling data.\n\n");
  while (threenext.input(datafile, TAGGEDFILE) != EOF) {
    /* determine closest cluster */
    vec.threeprev = threeprev;
    vec.twoprev = twoprev;
    vec.prev = prev;
    vec.curr = curr;
    vec.next = next;
    vec.twonext = twonext;
    vec.threenext = threenext;
    vec.lastcall = lastcall;
    vec.timetocall = time - timeofcall;
    vec.lastcallval = lastcallval;
    vec.normalize();
    call = vec.call(clust);

    /* determine if distance at previous time point is a local minimum */
    if (FULLOUTPUT) printf("\n%8.2f ", prevmin);
    if ((twoprevmin > prevmin) && (prevmin <= currmin)
//       && (prevmin < 300)
//       && (disttocall > (0.35*avedist(lastcall, prevcall, clust)))
         ){
      /* actual call - update call variables and print (Note: since distance
          must be a local minimum to call, won't ever have two adjacent calls,
          so if prev time is a call, the lastcall values at next time are those
          from prev time (i.e. current time can't be a call). */
      lastcall = prevcall;
      timeofcall = time-1;
      lastcallval = prevval;
      basenum++;
      if (!TAGFILE) printf("%c", inttocall(prevcall));
      if (FULLOUTPUT && TAGGEDFILE) printf(" [%c]", inttocall(prev.tag));
      prev.tag = prevcall;
//      printf(" %4d", timeofcall-1);
      if ((!TAGFILE) && (basenum%79 == 0) && (!FULLOUTPUT)) printf("\n");
      disttocall = 0;
    }
    else {
      if (FULLOUTPUT) printf(".");
      if (TAGGEDFILE && FULLOUTPUT) printf(" [%c]", inttocall(prev.tag));
      disttocall++;
    }
    if (FULLOUTPUT) printf(" (%c%c)", inttocall(clusterprevtag(prevcluster)),
            inttocall(clustercurrtag(prevcluster)));

    if (TAGFILE) prev.print(stdout, 1);

    /* move to next time */
    time++;
    prevcall = call;
    prevval = curr.fluorescence(call);
    threeprev = twoprev;
    twoprev = prev;
    prev = curr;
    curr = next;
    next = twonext;
    twonext = threenext;
    twoprevmin = prevmin;
    prevmin = currmin;
    prevcluster = currcluster;
  }

  printf("\n");

  fclose(datafile);
  exit(0);
}
```

```
#include "cluster.h"

/* Call data by closest cluster. */

extern double currmin;

int main(int argc, char *argv[]) {
  int time=0;          /* current time */
  int basenum=0;       /* number of bases called */
  int timeofcall;      /* time of last base call */
  int lastcall;        /* last base call */
  int i;
  char ctag[2];        /* for reading tag character */

  FILE *datafile, *clusterfile;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <cluster file> <data file>\n", argv[0]);
    exit(1);
  }

  if (!(clusterfile = fopen(argv[1], "r"))) {
    fprintf(stderr, "Can not open cluster file %s.\n", argv[1]);
    exit(1);
  }

  if (argc < 3) datafile = stdin;
  else {
    if (!(datafile = fopen(argv[2], "r"))) {
      fprintf(stderr, "Can not open data file %s.\n", argv[2]);
      exit(1);
    }
  }

  /* read in clusters - later, fix to have only cluster averages and
     check for premature end of cluster file */
  cluster *clust[NUMCLUSTS];  /* clusters */
  cluster *rv;

  /* read through, collecting calls */
  for (i = 0; i < NUMCLUSTS; i++) {  /* initialize */
    if ((rv = (cluster *) malloc(sizeof(cluster))) != NULL) {
      clust[i] = rv;
      clust[i]->average.input(clusterfile);
    }
    else {
```

```
    fprintf(stderr, "\nNot enough memory - Cluster %d\n", i);
    exit(1);
  }
}

datapoint threeprev, twoprev, twonext, threenext;
datapoint prev, curr, next; /* previous, current, and next data points */
vector vec;                 /* current vector */


char call;
double twoprevmin = 0, prevmin = 0;
int prevcall = -1, prevbase = 0;
int disttocall = 0;

/* initialize threeprev through twonext to first six time points */
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &threeprev.c, &threeprev.t, &threeprev.a, &threeprev.g, ctag);
threeprev.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &twoprev.c, &twoprev.t, &twoprev.a, &twoprev.g, ctag);
twoprev.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &prev.c, &prev.t, &prev.a, &prev.g, ctag);
prev.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &curr.c, &curr.t, &curr.a, &curr.g, ctag);
curr.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &next.c, &next.t, &next.a, &next.g, ctag);
next.tag = calltoint(ctag[0]);
fscanf(datafile, "%lf%lf%lf%lf%ls",
        &twonext.c, &twonext.t, &twonext.a, &twonext.g, ctag);
twonext.tag = calltoint(ctag[0]);
time = 2;
lastcall = 0;
timeofcall = 0;

/* read through */
fprintf(stderr, "Calling data.\n\n");
while ((fscanf(datafile, "%lf%lf%lf%lf%ls",
                 &threenext.c, &threenext.t, &threenext.a, &threenext.g, ctag))
        != EOF) {
  threenext.tag = calltoint(ctag[0]);

  /* determine closest cluster */
```

```
    vec.threeprev = threeprev;
    vec.prev = prev;
    vec.curr = curr;
    vec.next = next;
    vec.threenext = threenext;
    vec.tag = curr.tag;
    vec.lastcall = lastcall;
    vec.timetocall = time - timeofcall;
    vec.normalize();
    call = vec.call(clust);

    /* determine if distance is a local minimum */
 /*        printf("\n%15.6f ", prevmin); */
    if ((twoprevmin > prevmin) && (prevmin <= currmin)
//       && (prevmin < 20000)
         && (disttocall > (0.35*avedist(prevbase, prevcall, clust)))
         ){
      lastcall = prevcall;
      timeofcall = time-1;
      basenum++;
      printf("%c", inttocall(prevcall));
//      printf(" %4d\n", timeofcall-1);
      disttocall = 0;
      prevbase = prevcall;
    }
    else {
 /*      printf("."); */
      disttocall++;
    }

 /*      printf(" [%c]", inttocall(prev.tag)); */

    /* move to next time */
//    if (basenum%79 == 1) printf("\n");
    time++;
    threeprev = twoprev;
    twoprev = prev;
    prev = curr;
    curr = next;
    next = twonext;
    twonext = threenext;
    prevcall = call;
    twoprevmin = prevmin;
    prevmin = currmin;
  }
  printf("\n");

  fclose(datafile);
  exit(0);
}
```

```
/* Smooths out spikes.  Writes to standard output.  Reads from argument file
   if there is one, otherwise reads from standard input. */

extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
  #include <math.h>
}

#include "cluster.h"

main(int argc, char *argv[]) {
  FILE *infile;

  if (argc < 2) infile = stdin;
  else if (!(infile = fopen(argv[1], "r"))) {
    fprintf(stderr, "%s: can not open %s\n", argv[0], argv[1]);
    exit(1);
  }

  datapoint prev, curr, next;

  /* initialize prev and curr */
  prev.input(infile);
  curr.input(infile);
  printf("%10.0f %10.0f %10.0f %10.0f\n", prev.c, prev.t, prev.a, prev.g);

  while (next.input(infile) != EOF) {
    if ((fabs(curr.c-prev.c) > 20000) && (fabs(curr.c-next.c) > 20000) &&
        (fabs(next.c-prev.c) < 20000))
      curr.c = (prev.c + next.c) / 2.0;
    if ((fabs(curr.t-prev.t) > 20000) && (fabs(curr.t-next.t) > 20000) &&
        (fabs(next.t-prev.t) < 20000))
      curr.t = (prev.t + next.t) / 2.0;
    if ((fabs(curr.a-prev.a) > 20000) && (fabs(curr.a-next.a) > 20000) &&
        (fabs(next.a-prev.a) < 20000))
      curr.a = (prev.a + next.a) / 2.0;
    if ((fabs(curr.g-prev.g) > 20000) && (fabs(curr.g-next.g) > 20000) &&
        (fabs(next.g-prev.g) < 20000))
      curr.g = (prev.g + next.g) / 2.0;
    printf("%10.0f %10.0f %10.0f %10.0f\n", curr.c, curr.t, curr.a, curr.g);
    prev = curr;
    curr = next;
  }

  printf("%10.0f %10.0f %10.0f %10.0f\n", next.c, next.t, next.a, next.g);
```

```
    fclose(infile);

    exit(0);
  }
```

```
/* Background subtraction.  Takes four minima: current - BIGWIN to
   current, current - SMALLWIN to current, current to current +
   SMALLWIN, and current to current + BIGWIN.  Fits a line to these points
   using code from p. 527 of Numerical Recipes in C.  Evaluates this line
   at the current point and subtracts that value.  Does concurrently for four
   channels.  Writes to standard output.  If argument, reads input from
   argument file, otherwise reads from standard input.
*/

extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
  #include <math.h>
}

#include "linefit.h"

#define SMALLWIN 50
#define BIGWIN   100

int bufsize = 0;

class entry {
 public:
  double val;
  int time;
  entry *prevval;
  entry *nextval;
  entry *prevtime;
  entry *nexttime;
  entry(double v = 0, int t = 0) {val = v; time = t;};
  void init(double v = 0, int t = 0) {val = v; time = t;};
};

class window {
  entry *lowval = NULL;
  entry *lowtime = NULL;
  entry *hightime = NULL;
 public:
  void insert(entry *);
  entry *clearlowtime();
  void walkvals();
  void walktime();
  double min(){return lowval->val;};
  int mintime(){return lowval->time;};
};
```

```
class channel {
 public:
  window smallprevwin, bigprevwin, smallnextwin, bignextwin;
  double bigprevmin, smallprevmin, smallnextmin, bignextmin;
  int bigprevmintime, smallprevmintime, smallnextmintime, bignextmintime;
  int previndex, smallnextindex, bignextindex;
  int prevtime, smallnexttime, bignexttime;

  double buf[2*BIGWIN - 1];
  void walkalltimes();
  void walkallvals();
  void setallmins();
  void printallmins();
  void timezero();
  void firstphasestep();
  void secondphasestep();
  void mainphasestep();
  void endphaseonestep();
  void endphasetwostep();
  double fitcurrtime();
  void subtract();
};

class fourchannels {
 public:
  channel c, a, g, t;
  void bufinit(FILE *);
  void timezero();
  void firstphase();
  void secondphase();
  void mainphase(FILE *);
  void endphaseone();
  void endphasetwo();
};

void channel::subtract() {
 printf("%10.0f", buf[previndex] - fitcurrtime());
//  printf("%10.0f", fitcurrtime());  /* To get background */
}

double channel::fitcurrtime() {
//  walkalltimes();
  setallmins();
//  printallmins();
//  printf("Fitting line: ");
```

```
  /* if all are equal, return that value */
  if ((bigprevmintime == bignextmintime) &&
      (smallprevmintime == smallnextmintime) &&
      (bigprevmintime == smallprevmintime))
    {
//      printf("all points are equal\n");
//      printf("Subtraction value at %d is %.2f\n",
//            prevtime, bigprevmin);
      return (bigprevmin);
    }

  /* otherwise fit to a line */
  double x[4] = {double (bigprevmintime), double (smallprevmintime),
                     double (smallnextmintime), double (bignextmintime)};
  double y[4] = {bigprevmin, smallprevmin, smallnextmin, bignextmin};
  double *xx = x-1;
  double *yy = y-1;
  double sig[4];
  double a, b, siga, sigb, chi2, q;

  fit(xx, yy, 4, sig, 0, &a, &b, &siga, &sigb, &chi2, &q);

//  printf("y = %.2f + %.2f(x)\n", a, b);
//  printf("Subtraction value at %d is %.2f\n",
//        prevtime, a + b*(double(prevtime)));
   return (a + b*(double(prevtime)));
}

void fourchannels::endphaseone() {
  int i;

  for (i = 0; i < SMALLWIN; i++) {
    c.endphaseonestep();
    printf(" ");
    t.endphaseonestep();
    printf(" ");
    a.endphaseonestep();
    printf(" ");
    g.endphaseonestep();
    printf("\n");
  }
}

void channel::endphaseonestep() {
  /* scroll off ends */
  entry *curr;

  /* scroll all but bignextwin, drop low entry for bignextwin */
  curr = bigprevwin.clearlowtime();
  curr->init(buf[previndex], prevtime);
  bigprevwin.insert(curr);

  curr = smallprevwin.clearlowtime();
  curr->init(buf[previndex], prevtime);
  smallprevwin.insert(curr);

  curr = smallnextwin.clearlowtime();
  curr->init(buf[smallnextindex], smallnexttime);
  smallnextwin.insert(curr);

  curr = bignextwin.clearlowtime();
  delete curr;

  subtract();

  prevtime++;
  smallnexttime++;
  previndex = (previndex+1)%bufsize;
  smallnextindex = (smallnextindex+1)%bufsize;
}

void fourchannels::endphasetwo() {
  int i;

  for (i = SMALLWIN; i < BIGWIN - 1; i++) {
    c.endphasetwostep();
    printf(" ");
    t.endphasetwostep();
    printf(" ");
    a.endphasetwostep();
    printf(" ");
    g.endphasetwostep();
    printf("\n");
  }
}

void channel::endphasetwostep() {
  entry *curr;

  /* scroll previous windows, drop low entry for nextwindows */
  curr = bigprevwin.clearlowtime();
  curr->init(buf[previndex], prevtime);
  bigprevwin.insert(curr);

  curr = smallprevwin.clearlowtime();
  curr->init(buf[previndex], prevtime);
  smallprevwin.insert(curr);

  curr = smallnextwin.clearlowtime();
  delete curr;
  curr = bignextwin.clearlowtime();
  delete curr;

  subtract();

  prevtime++;
  previndex = (previndex+1)%bufsize;
}

void fourchannels::mainphase(FILE *infile) {
  int i;
  entry *curr;

  /* all windows are full - continue moving current time forward */
  while ((fscanf(infile, "%lf%lf%lf%lf",
                   &c.buf[c.bignextindex], &t.buf[t.bignextindex],
                   &a.buf[a.bignextindex], &g.buf[g.bignextindex]) != EOF)) {
    /* scroll all windows */
    c.mainphasestep();
    printf(" ");
    t.mainphasestep();
    printf(" ");
    a.mainphasestep();
    printf(" ");
    g.mainphasestep();
    printf("\n");
  }
}

void channel::mainphasestep() {
  int i;
  entry *curr;

  /* scroll all windows */
  curr = bigprevwin.clearlowtime();
  curr->init(buf[previndex], prevtime);
  bigprevwin.insert(curr);
  curr = smallprevwin.clearlowtime();
  curr->init(buf[previndex], prevtime);
  smallprevwin.insert(curr);

  curr = smallnextwin.clearlowtime();
  curr->init(buf[smallnextindex], smallnexttime);
  smallnextwin.insert(curr);

  curr = bignextwin.clearlowtime();
  curr->init(buf[bignextindex], bignexttime);
  bignextwin.insert(curr);

  subtract();

  previndex = (previndex+1)%bufsize;
  smallnextindex = (smallnextindex+1)%bufsize;
  bignextindex = (bignextindex+1)%bufsize;
  prevtime++;
  smallnexttime++;
  bignexttime++;
}

void fourchannels::secondphase() {
  int i;

  for (i = BIGWIN + SMALLWIN - 1; i < 2*BIGWIN - 1; i++) {
    c.secondphasestep();
    printf(" ");
    t.secondphasestep();
    printf(" ");
    a.secondphasestep();
    printf(" ");
    g.secondphasestep();
    printf("\n");
  }
}

void channel::secondphasestep() {
  entry *curr;

  /* bigprevwin is still not full - continue moving current time forward */
  /* insert into bigprev window, scroll other windows */
  curr = new entry(buf[previndex], prevtime);
  bigprevwin.insert(curr);
```

```
    curr = smallprevwin.clearlowtime();
    curr->init(buf[previndex], prevtime);
    smallprevwin.insert(curr);

    curr = smallnextwin.clearlowtime();
    curr->init(buf[smallnextindex], smallnexttime);
    smallnextwin.insert(curr);

    curr = bignextwin.clearlowtime();
    curr->init(buf[bignextindex], bignexttime);
    bignextwin.insert(curr);

    subtract();

    previndex = (previndex+1)%bufsize;
    smallnextindex = (smallnextindex+1)%bufsize;
    bignextindex = (bignextindex+1)%bufsize;
    prevtime++;
    smallnexttime++;
    bignexttime++;
  }

  void fourchannels::firstphase() {
    /* move current time forward - still have incomplete prev windows */
    int i;

    for (i = BIGWIN; i < BIGWIN + SMALLWIN - 1; i++) {
      c.firstphasestep();
      printf(" ");
      t.firstphasestep();
      printf(" ");
      a.firstphasestep();
      printf(" ");
      g.firstphasestep();
      printf("\n");
    }
  }

  void channel::firstphasestep() {
    entry *curr;

    /* insert into previous windows, scroll next windows */
    curr = new entry(buf[previndex], prevtime);
    bigprevwin.insert(curr);

    curr = new entry(buf[previndex], prevtime);
  smallprevwin.insert(curr);

  curr = smallnextwin.clearlowtime();
  curr->init(buf[smallnextindex], smallnexttime);
  smallnextwin.insert(curr);

  curr = bignextwin.clearlowtime();
  curr->init(buf[bignextindex], bignexttime);
  bignextwin.insert(curr);

  subtract();

  previndex = (previndex+1)%bufsize;
  smallnextindex = (smallnextindex+1)%bufsize;
  bignextindex = (bignextindex+1)%bufsize;
  prevtime++;
  smallnexttime++;
  bignexttime++;
}

void channel::timezero() {
  entry *curr;

  /* initialize to current time 0 */
  previndex = prevtime = 0;
  smallnextindex = smallnexttime = 0;
  bignextindex = bignexttime = 0;

  /* insert entry 0 into all windows */
  curr = new entry(buf[previndex], previndex);
  bigprevwin.insert(curr);

  curr = new entry(buf[previndex], previndex);
  smallprevwin.insert(curr);

  curr = new entry(buf[smallnextindex], smallnextindex);
  smallnextwin.insert(curr);

  curr = new entry(buf[bignextindex], bignextindex);
  bignextwin.insert(curr);

  previndex = (previndex+1)%bufsize;
  smallnextindex = (smallnextindex+1)%bufsize;
  bignextindex = (bignextindex+1)%bufsize;
  prevtime++;
  smallnexttime++;
  bignexttime++;

  int i;

  for (i = 1; i < SMALLWIN; i++) {
    /* insert into smallnextwin and bignextwin*/
    curr = new entry(buf[smallnextindex], smallnextindex);
    smallnextwin.insert(curr);

    curr = new entry(buf[bignextindex], bignextindex);
    bignextwin.insert(curr);

    smallnextindex = (smallnextindex+1)%bufsize;
    bignextindex = (bignextindex+1)%bufsize;
    smallnexttime++;
    bignexttime++;
  }
  for (i = SMALLWIN; i < BIGWIN; i++) {
    /* insert into bignextwin */
    curr = new entry(buf[bignextindex], bignextindex);
    bignextwin.insert(curr);

    bignextindex = (bignextindex+1)%bufsize;
    bignexttime++;
  }

  subtract();
}

void fourchannels::timezero() {
  c.timezero();
  printf(" ");
  t.timezero();
  printf(" ");
  a.timezero();
  printf(" ");
  g.timezero();
  printf("\n");
}

void fourchannels::bufinit(FILE *infile) {
  /* read in buffers */
  while ((bufsize < 2*BIGWIN - 1) &&
          (fscanf(infile, "%lf%lf%lf%lf",
                  &c.buf[bufsize], &t.buf[bufsize],
                  &a.buf[bufsize], &g.buf[bufsize]) != EOF))
    bufsize++;

  if (bufsize < 2*BIGWIN - 1) {
    fprintf(stderr, "Input too short for window size %d\n", BIGWIN);
    exit(1);
  }
}

void window::insert(entry *e) {
  entry *p;

  /* insert in value list */
  p = lowval;

  if (p == NULL) { /* list is empty - make first element */
    e->prevval = NULL;
    e->nextval = NULL;
    lowval = e;
  }
  else {
    while ((p->val < e->val) && (p->nextval != NULL)) p = p->nextval;
    if (e->val <= p->val) { /* insert before p */
      e->prevval = p->prevval;
      p->prevval = e;
      e->nextval = p;
      if (e->prevval != NULL) e->prevval->nextval = e;
      if (e->val <= lowval->val) lowval = e;
    }
    else { /* reached end of list - insert at end */
      e->prevval = p;
      p->nextval = e;
      e->nextval = NULL;
    }
  }

  /* insert in time list */
  if (lowtime == NULL) {  /* list is empty - make first element */
    e->prevtime = NULL;
    e->nexttime = NULL;
    lowtime = e;
    hightime = e;
  }
  else {
    e->prevtime = hightime;
    if (e->prevtime != NULL) e->prevtime->nexttime = e;
    e->nexttime = NULL;
    hightime = e;
  }
};

entry *window::clearlowtime() {
  /* removes low time entry from the list, returns pointer
     to entry for reuse */
  entry *temp;

  temp = lowtime;
  if (lowtime == NULL) return NULL;
  if (lowval == lowtime) lowval = lowval->nextval;
  if (hightime == lowtime) hightime = NULL;
  lowtime = lowtime->nexttime;
  if (lowtime != NULL) lowtime->prevtime = NULL;
  if (temp->nextval != NULL) temp->nextval->prevval = temp->prevval;
  if (temp->prevval != NULL) temp->prevval->nextval = temp->nextval;

  return(temp);
};

void window::walkvals() {
  entry *temp = lowval;

  while (temp != NULL) {
    printf("%.0f ", temp->val);
    temp = temp->nextval;
  }
  printf("\n");
};

void window::walktime() {
  entry *temp = lowtime;

  while (temp != NULL) {
    printf("%.0f ", temp->val);
    temp = temp->nexttime;
  }
  printf("\n");
};

void channel::walkalltimes() {
  bigprevwin.walktime();
  smallprevwin.walktime();
  smallnextwin.walktime();
  bignextwin.walktime();
  printf("\n");
}

void channel::walkallvals() {
  bigprevwin.walkvals();
  smallprevwin.walkvals();
  smallnextwin.walkvals();
  bignextwin.walkvals();
  printf("\n");
}

void channel::setallmins() {
  bigprevmin = bigprevwin.min();
  smallprevmin = smallprevwin.min();
  smallnextmin = smallnextwin.min();
  bignextmin = bignextwin.min();
  bigprevmintime = bigprevwin.mintime();
  smallprevmintime = smallprevwin.mintime();
  smallnextmintime = smallnextwin.mintime();
  bignextmintime = bignextwin.mintime();
}

void channel::printallmins() {
  printf("\nMins: (%d, %.0f) (%d, %.0f) (%d, %.0f) (%d, %.0f)\n",
          bigprevmintime, bigprevmin, smallprevmintime, smallprevmin,
          smallnextmintime, smallnextmin, bignextmintime, bignextmin);
}

main(int argc, char *argv[]) {
  FILE *infile;

//  malloc_debug(8);

  /* read from standard input if no arguments, otherwise read from
     argument file */
  if (argc < 2) infile = stdin;
  else if (!(infile = fopen(argv[1], "r"))) {
    fprintf(stderr, "%s: can not open file %s\n", argv[0], argv[1]);
    exit(1);
  }

  entry *curr;
  fourchannels chan;

  chan.bufinit(infile);
  chan.timezero();
  chan.firstphase();
  chan.secondphase();
  chan.mainphase(infile);
  chan.endphaseone();
  chan.endphasetwo();

  fclose(infile);

  exit(0);
}
```

```
/* Background subtraction.  Takes four minima: current - 200 to current - 100,
   current - 100 to current, current to current + 100, and current + 100 to
   current + 200.  Fits a line to these points using code from p. 527
   of Numerical Recipes in C.  Evaluates this line at the current point
   and subtracts that value.  Writes to standard output.  If argument,
   reads input from argument file, otherwise reads from standard input.
*/

extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
  #include <math.h>
}

#include <linefit.h>

#define WINSIZE 50

class entry {
 public:
  double val;
  int time;
  entry *prevval;
  entry *nextval;
  entry *prevtime;
  entry *nexttime;
  entry(double v = 0, int t = 0) {val = v; time = t;};
  void init(double v = 0, int t = 0) {val = v; time = t;};
};

class window {
  entry *lowval = NULL;
  entry *lowtime = NULL;
  entry *hightime = NULL;
 public:
  void insert(entry *);
  entry *clearlowtime();
  void walkvals();
  void walktime();
  double min(){return lowval->val;};
  int mintime(){return lowval->time;};
};

class channel {
 public:
  window closeprevwin, farprevwin, closenextwin, farnextwin;
  double farprevmin, closeprevmin, closenextmin, farnextmin;
  int farprevmintime, closeprevmintime, closenextmintime, farnextmintime;
  int farprevindex, closeprevindex, closenextindex, farnextindex;
  int farprevtime, closeprevtime, closenexttime, farnexttime;

  double buf[4*WINSIZE - 1];
  int bufsize = 0;
  void walkalltimes();
  void walkallvals();
  void setallmins();
  void printallmins();
  void bufinit(FILE *);
  void timezero();
  void firstphase();
  void secondphase();
  void mainphase(FILE *);
  void endphase();
  double fitcurrtime();
  void subtract();
};

void channel::subtract() {
 printf("%.0f\n", buf[closeprevindex] - fitcurrtime());
//  printf("%.0f\n", fitcurrtime());  /* To get background */
}

double channel::fitcurrtime() {
//  walkalltimes();
  setallmins();
//  printallmins();
//  printf("Fitting line: ");

  /* if all are equal, return that value */
  if ((farprevmintime == farnextmintime) &&
      (closeprevmintime == closenextmintime) &&
      (farprevmintime == closeprevmintime))
    {
//      printf("all points are equal\n");
//      printf("Subtraction value at %d is %.2f\n",
//             farprevtime, farprevmin);
      return (farprevmin);
    }

  /* otherwise fit to a line */
  double x[4] = {double (farprevmintime), double (closeprevmintime),
                 double (closenextmintime), double (farnextmintime)};
  double y[4] = {farprevmin, closeprevmin, closenextmin, farnextmin};
  double *xx = x-1;
  double *yy = y-1;
  double sig[4];
  double a, b, siga, sigb, chi2, q;

  fit(xx, yy, 4, sig, 0, &a, &b, &siga, &sigb, &chi2, &q);

//  printf("y = %.2f + %.2f(x)\n", a, b);
//  printf("Subtraction value at %d is %.2f\n",
//        closeprevtime, a + b*(double(closeprevtime)));
   return (a + b*(double(closeprevtime)));
}

void channel::endphase() {
  /* scroll off ends */
  int i;
  entry *curr;

  for (i = 0; i < WINSIZE; i++) {
    /* scroll all but farnextwin */
    curr = farprevwin.clearlowtime();
    curr->init(buf[farprevindex], farprevtime);
    farprevwin.insert(curr);

    curr = closeprevwin.clearlowtime();
    curr->init(buf[closeprevindex], closeprevtime);
    closeprevwin.insert(curr);

    curr = closenextwin.clearlowtime();
    curr->init(buf[closenextindex], closenexttime);
    closenextwin.insert(curr);

    subtract();

    farprevtime++;
    closeprevtime++;
    closenexttime++;
    farprevindex = (farprevindex+1)%bufsize;
    closeprevindex = (closeprevindex+1)%bufsize;
    closenextindex = (closenextindex+1)%bufsize;
  }

  for (i = 1; i < WINSIZE; i++) {
    /* scroll previous windows, drop low entry for nextwindows */
    curr = farprevwin.clearlowtime();
    curr->init(buf[farprevindex], farprevtime);
    farprevwin.insert(curr);

    curr = closeprevwin.clearlowtime();
    curr->init(buf[closeprevindex], closeprevtime);
    closeprevwin.insert(curr);

    curr = closenextwin.clearlowtime();
    delete curr;
    curr = farnextwin.clearlowtime();
    delete curr;

    subtract();

    farprevtime++;
    closeprevtime++;
    farprevindex = (farprevindex+1)%bufsize;
    closeprevindex = (closeprevindex+1)%bufsize;
  }
}

void channel::mainphase(FILE *infile) {
  int i;
  entry *curr;

  /* all windows are full - continue moving current time forward */
  while ((fscanf(infile, "%lf", &buf[farnextindex]) == 1)) {
    /* scroll all windows */
    curr = farprevwin.clearlowtime();
    curr->init(buf[farprevindex], farprevtime);
    farprevwin.insert(curr);

    curr = closeprevwin.clearlowtime();
    curr->init(buf[closeprevindex], closeprevtime);
    closeprevwin.insert(curr);

    curr = closenextwin.clearlowtime();
    curr->init(buf[closenextindex], closenexttime);
    closenextwin.insert(curr);

    curr = farnextwin.clearlowtime();
    curr->init(buf[farnextindex], farnexttime);
    farnextwin.insert(curr);

    subtract();
    farprevindex = (farprevindex+1)%bufsize;
    closeprevindex = (closeprevindex+1)%bufsize;
    closenextindex = (closenextindex+1)%bufsize;
    farnextindex = (farnextindex+1)%bufsize;
    farprevtime++;
    closeprevtime++;
    closenexttime++;
    farnexttime++;
  }
}

void channel::secondphase() {
  int i;
  entry *curr;

  for (i = 0; i < WINSIZE; i++) {
    /* leave farprev window alone, scroll other windows */

    curr = closeprevwin.clearlowtime();
    curr->init(buf[closeprevindex], closeprevtime);
    closeprevwin.insert(curr);

    curr = closenextwin.clearlowtime();
    curr->init(buf[closenextindex], closenexttime);
    closenextwin.insert(curr);

    curr = farnextwin.clearlowtime();
    curr->init(buf[farnextindex], farnexttime);
    farnextwin.insert(curr);

    subtract();

    closeprevindex = (closeprevindex+1)%bufsize;
    closenextindex = (closenextindex+1)%bufsize;
    farnextindex = (farnextindex+1)%bufsize;
    closeprevtime++;
    closenexttime++;
    farnexttime++;
  }
}

void channel::firstphase() {
  /* move current time forward - still have incomplete prev windows */
  int i;
  entry *curr;
  for (i = 1; i < WINSIZE; i++) {
    /* insert into previous windows, scroll next windows */
    curr = new entry(buf[farprevindex], farprevtime);
    farprevwin.insert(curr);

    curr = new entry(buf[closeprevindex], closeprevtime);
    closeprevwin.insert(curr);

    curr = closenextwin.clearlowtime();
    curr->init(buf[closenextindex], closenexttime);
    closenextwin.insert(curr);

    curr = farnextwin.clearlowtime();
    curr->init(buf[farnextindex], farnexttime);
    farnextwin.insert(curr);

    subtract();

    farprevindex = (farprevindex+1)%bufsize;
    closeprevindex = (closeprevindex+1)%bufsize;
    closenextindex = (closenextindex+1)%bufsize;
    farnextindex = (farnextindex+1)%bufsize;
    farprevtime++;
    closeprevtime++;
    closenexttime++;
    farnexttime++;
  }
}

void channel::timezero() {
  entry *curr;

  /* initialize to current time 0 */
  farprevindex = farprevtime = 0;
  closeprevindex = closeprevtime = 0;
  closenextindex = closenexttime = 0;
  farnextindex = farnexttime = WINSIZE;

  /* insert entry 0 into prev windows */
  curr = new entry(buf[farprevindex], farprevindex);
  farprevwin.insert(curr);

  curr = new entry(buf[closeprevindex], closeprevindex);
  closeprevwin.insert(curr);

  farprevindex = (farprevindex+1)%bufsize;
  closeprevindex = (closeprevindex+1)%bufsize;
  farprevtime++;
  closeprevtime++;

  int i;

  /* fill next windows */
  for (i = 0; i < WINSIZE; i++) {
    /* insert into closenextwin and farnextwin*/
    curr = new entry(buf[closenextindex], closenextindex);
    closenextwin.insert(curr);

    curr = new entry(buf[farnextindex], farnextindex);
    farnextwin.insert(curr);

    closenextindex = (closenextindex+1)%bufsize;
    farnextindex = (farnextindex+1)%bufsize;
    closenexttime++;
    farnexttime++;
  }

  subtract();
}

void channel::bufinit(FILE *infile) {
  /* read in buffer */
  while ((bufsize < 4*WINSIZE - 1) &&
          (fscanf(infile, "%lf", &buf[bufsize]) == 1))
    bufsize++;

  if (bufsize < 4*WINSIZE - 1) {
    fprintf(stderr, "Input too short for window size %d\n", WINSIZE);
    exit(1);
  }
}

void window::insert(entry *e) {
  entry *p;

  /* insert in value list */
  p = lowval;

  if (p == NULL) { /* list is empty - make first element */
    e->prevval = NULL;
    e->nextval = NULL;
    lowval = e;
  }
  else {
    while ((p->val < e->val) && (p->nextval != NULL)) p = p->nextval;
    if (e->val <= p->val) { /* insert before p */
      e->prevval = p->prevval;
      p->prevval = e;
      e->nextval = p;
      if (e->prevval != NULL) e->prevval->nextval = e;
      if (e->val <= lowval->val) lowval = e;
    }
    else { /* reached end of list - insert at end */
      e->prevval = p;
      p->nextval = e;
      e->nextval = NULL;
    }
  }

  /* insert in time list */
  if (lowtime == NULL) {  /* list is empty - make first element */
    e->prevtime = NULL;
    e->nexttime = NULL;
    lowtime = e;
    hightime = e;
  }
  else {
    e->prevtime = hightime;
    if (e->prevtime != NULL) e->prevtime->nexttime = e;
    e->nexttime = NULL;
    hightime = e;
  }
};

entry *window::clearlowtime() {
  /* removes low time entry from the list, returns pointer
     to entry for reuse */
  entry *temp;

  temp = lowtime;
  if (lowtime == NULL) return NULL;
  if (lowval == lowtime) lowval = lowval->nextval;
  if (hightime == lowtime) hightime = NULL;
  lowtime = lowtime->nexttime;
  if (lowtime != NULL) lowtime->prevtime = NULL;
  if (temp->nextval != NULL) temp->nextval->prevval = temp->prevval;
  if (temp->prevval != NULL) temp->prevval->nextval = temp->nextval;
  return(temp);
};

void window::walkvals() {
  entry *temp = lowval;

  while (temp != NULL) {
    printf("%.0f ", temp->val);
    temp = temp->nextval;
  }
  printf("\n");
};

void window::walktime() {
  entry *temp = lowtime;

  while (temp != NULL) {
    printf("%.0f ", temp->val);
    temp = temp->nexttime;
  }
  printf("\n");
};

void channel::walkalltimes() {
  farprevwin.walktime();
  closeprevwin.walktime();
  closenextwin.walktime();
  farnextwin.walktime();
  printf("\n");
}

void channel::walkallvals() {
  farprevwin.walkvals();
  closeprevwin.walkvals();
  closenextwin.walkvals();
  farnextwin.walkvals();
  printf("\n");
}

void channel::setallmins() {
  farprevmin = farprevwin.min();
  closeprevmin = closeprevwin.min();
  closenextmin = closenextwin.min();
  farnextmin = farnextwin.min();
  farprevmintime = farprevwin.mintime();
  closeprevmintime = closeprevwin.mintime();
```

```
    closenextmintime = closenextwin.mintime();
    farnextmintime = farnextwin.mintime();
  }

  void channel::printallmins() {
    printf("\nMins: (%d, %.0f) (%d, %.0f) (%d, %.0f) (%d, %.0f)\n",
            farprevmintime, farprevmin, closeprevmintime, closeprevmin,
            closenextmintime, closenextmin, farnextmintime, farnextmin);
  }

  main(int argc, char *argv[]) {
    FILE *infile;

  //  malloc_debug(8);

    /* read from standard input if no arguments, otherwise read from
       argument file */
    if (argc < 2) infile = stdin;
    else if (!(infile = fopen(argv[1], "r"))) {
      fprintf(stderr, "%s: can not open file %s\n", argv[0], argv[1]);
      exit(1);
    }

    entry *curr;
    channel c;

    c.bufinit(infile);
    c.timezero();
    c.firstphase();
    c.secondphase();
    c.mainphase(infile);
    c.endphase();

    fclose(infile);

    exit(0);
  }
```

```
/* Contains member functions for cluster and vector classes, as well as
   other auxilliary functions. */

#include "cluster.h"

extern double currmin;
extern int currcluster;

int round(double x) {
  return ((x - int(x) > .5) ? (int(x)+1) : int(x));
}

void datapoint::print(FILE *fp, int flag) {
  /* if flag is one, print tag; if zero don't print tag */
  fprintf(fp, "%10.6f ", c);
  fprintf(fp, "%10.6f ", t);
  fprintf(fp, "%10.6f ", a);
  fprintf(fp, "%10.6f", g);
  if (flag) fprintf(fp, "%6c", inttocall(tag));
  fprintf(fp, "\n");
}

void vector::print(FILE *fp) {
  threeprev.print(fp);
  twoprev.print(fp);
  prev.print(fp);
  curr.print(fp);
  next.print(fp);
  twonext.print(fp);
  threenext.print(fp);
  fprintf(fp, "%3d %5.1f %10.6f %10.6f %10.6f %10.6f %3d\n",
          lastcall, timetocall, lastcallval, max, twoprevmax, twonextmax,
          tag);
}

int datapoint::input(FILE *fp, int flag) {
  /* read in from file.  If flag is 0, don't look for tag. Sets all negative
     values to 0. If flag is 1, looks for alphabetic tags, if 2, looks for
     numeric tags. */
  int rv;
  char ctag[2];

  rv = fscanf(fp, "%lf %lf %lf %lf", &c, &t, &a, &g);
  if ((flag == 1) && (rv != EOF)) {
    rv = fscanf(fp, "%1s", ctag);
    tag = calltoint(ctag[0]);
  }
  if ((flag == 2) && (rv != EOF)) rv = fscanf(fp, "%d", &tag);
  if (flag == 0) tag = -1;

  if (c < 0) c = 0;
  if (t < 0) t = 0;
  if (a < 0) a = 0;
  if (g < 0) g = 0;

  return rv;
}

double datapoint::fluorescence(int tag) {
  switch (tag) {
  case 0:
    return c;
    break;
  case 1:
    return t;
    break;
  case 2:
    return a;
    break;
  case 3:
    return g;
    break;
  default:
    return 0;
    break;
  }
}

void vector::input(FILE *fp) {
  /* read in from file */
  threeprev.input(fp);
  twoprev.input(fp);
  prev.input(fp);
  curr.input(fp);
  next.input(fp);
  twonext.input(fp);
  threenext.input(fp);
  fscanf(fp, "%d %lf %lf %lf %lf %lf %d\n",
          &lastcall, &timetocall, &lastcallval,
          &max, &twoprevmax, &twonextmax, &tag);
}
```

```
void vector::normalize() {
  /* normalize all fluorescence values so that max value is 100, and set
     max field to absolute max.  */

  double max1, max2, max3, max4, max5, max6, max7, max8, max9, max10;
  /* quick hack! */

  /* round one - compare pairs of values */
  max1 = (prev.c > prev.a)? prev.c : prev.a;
  max2 = (prev.g > prev.t)? prev.g : prev.t;
  max3 = (curr.c > curr.a)? curr.c : curr.a;
  max4 = (curr.g > curr.t)? curr.g : curr.t;
  max5 = (next.c > next.a)? next.c : next.a;
  max6 = (next.g > next.t)? next.g : next.t;
  max7 = (twoprev.c > twoprev.a)? twoprev.c : twoprev.a;
  max8 = (twoprev.g > twoprev.t)? twoprev.g : twoprev.t;
  max9 = (twonext.c > twonext.a)? twonext.c : twonext.a;
  max10 = (twonext.g > twonext.t)? twonext.g : twonext.t;

  /* round two - compare winners of round 1 */
  max1 = (max1 > max2)? max1 : max2;
  max2 = (max3 > max4)? max3 : max4;
  max3 = (max5 > max6)? max5 : max6;
  max4 = (max7 > max8)? max7 : max8;
  twoprevmax = max4;
  max5 = (max9 > max10)? max9 : max10;
  twonextmax = max5;

  /* round three - determine largest winner of round 2 */
  max1 = (max1 > max2)? max1 : max2;
  max2 = (max3 > max4)? max3 : max4;
  max3 = max5;

  /* round four - determine largest winner of round 3 */
  max1 = (max1 > max2)? max1 : max2;
  max1 = (max1 > max3)? max1 : max3;
  max = max1; /* set max field */

  /* max1 is now max.  Normalize values */
  if (max1 != 0) {
    threeprev.c = threeprev.c / max1;
    threeprev.a = threeprev.a / max1;
    threeprev.g = threeprev.g / max1;
    threeprev.t = threeprev.t / max1;
    twoprev.c = twopr v.c / max1;
    twoprev.a = twoprev.a / max1;
```

```
      twoprev.g = twoprev.g / max1;
      twoprev.t = twoprev.t / max1;
      prev.c = prev.c / max1;
      prev.a = prev.a / max1;
      prev.g = prev.g / max1;
      prev.t = prev.t / max1;
      curr.c = curr.c / max1;
      curr.a = curr.a / max1;
      curr.g = curr.g / max1;
      curr.t = curr.t / max1;
      next.c = next.c / max1;
      next.a = next.a / max1;
      next.g = next.g / max1;
      next.t = next.t / max1;
      twonext.c = twonext.c / max1;
      twonext.a = twonext.a / max1;
      twonext.g = twonext.g / max1;
      twonext.t = twonext.t / max1;
      threenext.c = threenext.c / max1;
      threenext.a = threenext.a / max1;
      threenext.g = threenext.g / max1;
      threenext.t = threenext.t / max1;
    }
}

int iscall(int call) {
  /* determines whether a data line has been called */
  return((call >= 0) && (call <= 4));
}

int calltoint(char call) {
  /* converts character tag to int */
  switch (call) {
  case 'C':
    return 0;
  case 'A':
    return 1;
  case 'G':
    return 2;
  case 'T':
    return 3;
  case 'X':
  default:
    return -1;
  }
}
```

```
char inttocall(int tag) {
  switch (tag) {
  case 0:
    return ’C’;
  case 1:
    return ’A’;
  case 2:
    return ’G’;
  case 3:
    return ’T’;
  case -1:
  default:
    return ’X’;
  }
}

void cluster:: addvec(vector *vec) {
  /* adds the vector vec to the cluster clust */
  if (vec->max != 0) {
    if (size >= 64) {
      fprintf(stderr, "Warning: Too many vectors in cluster %c %c\n",
               inttocall(vec->lastcall),inttocall(vec->tag));
    }
    else {
      data[size] = *vec;
      size++;
    }
  }
}

void cluster:: create_average() {
  /* creates arithmetic average vector from vectors in data array */
  int i;
  double tpcs = 0.0, tpas = 0.0, tpgs = 0.0, tpts = 0.0;
  double twpcs = 0.0, twpas = 0.0, twpgs = 0.0, twpts = 0.0;
  double pcs = 0.0, pas = 0.0, pgs = 0.0, pts = 0.0;
  double ccs = 0.0, cas = 0.0, cgs = 0.0, cts = 0.0;
  double ncs = 0.0, nas = 0.0, ngs = 0.0, nts = 0.0;
  double twncs = 0.0, twnas = 0.0, twngs = 0.0, twnts = 0.0;
  double tncs = 0.0, tnas = 0.0, tngs = 0.0, tnts = 0.0;
  double ttcs = 0.0, lcvs = 0.0, ms = 0.0, pms = 0.0, nms = 0.0;
  vector *cp;

  /* sum up each vector component */
  if (size == 0) fprintf(stderr, "No data points in this cluster!\n");
```

```
//  fprintf(stderr, "\nCluster %c%c:\n     ", inttocall(data[0].lastcall),
//         inttocall(data[0].tag));

  for (i = 0; i < size; i++) {
    cp = &data[i];
//    if (i>0) fprintf(stderr, "%d ", cp->timetocall);
    tpcs += cp->threeprev.c;
    tpas += cp->threeprev.a;
    tpgs += cp->threeprev.g;
    tpts += cp->threeprev.t;
    twpcs += cp->twoprev.c;
    twpas += cp->twoprev.a;
    twpgs += cp->twoprev.g;
    twpts += cp->twoprev.t;
    pcs += cp->prev.c;
    pas += cp->prev.a;
    pgs += cp->prev.g;
    pts += cp->prev.t;
    ccs += cp->curr.c;
    cas += cp->curr.a;
    cgs += cp->curr.g;
    cts += cp->curr.t;
    ncs += cp->next.c;
    nas += cp->next.a;
    ngs += cp->next.g;
    nts += cp->next.t;
    twncs += cp->twonext.c;
    twnas += cp->twonext.a;
    twngs += cp->twonext.g;
    twnts += cp->twonext.t;
    tncs += cp->threenext.c;
    tnas += cp->threenext.a;
    tngs += cp->threenext.g;
    tnts += cp->threenext.t;
    if (i>0) {  /* ignore timetocall and lastcallval for first data point,
                   since may be bad */
      ttcs += cp->timetocall;
      lcvs += cp->lastcallval;
    }
    ms += cp->max;
    pms += cp->twoprevmax;
    nms += cp->twonextmax;
  }

  /* take averages */
  average.threeprev.c = tpcs/size;
  average.threeprev.a = tpas/size;
  average.threeprev.g = tpgs/size;
  average.threeprev.t = tpts/size;
  average.twoprev.c = twpcs/size;
  average.twoprev.a = twpas/size;
  average.twoprev.g = twpgs/size;
  average.twoprev.t = twpts/size;
  average.prev.c = pcs/size;
  average.prev.a = pas/size;
  average.prev.g = pgs/size;
  average.prev.t = pts/size;
  average.curr.c = ccs/size;
  average.curr.a = cas/size;
  average.curr.g = cgs/size;
  average.curr.t = cts/size;
  average.next.c = ncs/size;
  average.next.a = nas/size;
  average.next.g = ngs/size;
  average.next.t = nts/size;
  average.twonext.c = twncs/size;
  average.twonext.a = twnas/size;
  average.twonext.g = twngs/size;
  average.twonext.t = twnts/size;
  average.threenext.c = tncs/size;
  average.threenext.a = tnas/size;
  average.threenext.g = tngs/size;
  average.threenext.t = tnts/size;
  average.timetocall = ttcs/(size-1);
  average.lastcallval = lcvs/(size-1);
  average.max = ms/size;
  average.twoprevmax = pms/size;
  average.twonextmax = nms/size;

  average.lastcall = data[0].lastcall;  /* same for whole cluster */
  average.tag = data[0].curr.tag;       /* same for whole cluster */

//  fprintf(stderr, "(average %.2f)", average.timetocall);

}

double cluster:: distance(vector *vec) {
  double dist, vmax, amax, vlast, alast;

  if (vec->max <= 0) vmax = 1;
  else vmax = vec->max;
  if (average.max <= 0) amax = 1;
  else amax = average.max;
  if (vec->lastcallval <= 0) vlast = 1;
  else vlast = vec->lastcallval;
  if (average.lastcallval <= 0) alast = 1;
  else alast = average.lastcallval;


  dist = (
    (
/*     pow((vec->threeprev.c - average.threeprev.c), 2) +
     pow((vec->threeprev.a - average.threeprev.a), 2) +
     pow((vec->threeprev.g - average.threeprev.g), 2) +
     pow((vec->threeprev.t - average.threeprev.t), 2) +
*/     pow((vec->twoprev.c - average.twoprev.c), 2) +
     pow((vec->twoprev.a - average.twoprev.a), 2) +
     pow((vec->twoprev.g - average.twoprev.g), 2) +
     pow((vec->twoprev.t - average.twoprev.t), 2) +
     pow((vec->prev.c - average.prev.c)*1.5, 2) +
     pow((vec->prev.a - average.prev.a)*1.5, 2) +
     pow((vec->prev.g - average.prev.g)*1.5, 2) +
     pow((vec->prev.t - average.prev.t)*1.5, 2) +
     pow((vec->curr.c - average.curr.c)*2.0, 2) +
     pow((vec->curr.a - average.curr.a)*2.0, 2) +
     pow((vec->curr.g - average.curr.g)*2.0, 2) +
     pow((vec->curr.t - average.curr.t)*2.0, 2) +
     pow((vec->next.c - average.next.c)*1.5, 2) +
     pow((vec->next.a - average.next.a)*1.5, 2) +
     pow((vec->next.g - average.next.g)*1.5, 2) +
     pow((vec->next.t - average.next.t)*1.5, 2) +
/*     pow(vec->prev.c - average.prev.c + vec->next.c - average.next.c, 2) +
     pow(vec->prev.a - average.prev.a + vec->next.a - average.next.a, 2) +
     pow(vec->prev.g - average.prev.g + vec->next.g - average.next.g, 2) +
     pow(vec->prev.t - average.prev.t + vec->next.t - average.next.t, 2) +
     pow(vec->twoprev.c - average.twoprev.c +
          vec->twonext.c - average.twonext.c, 2) +
     pow(vec->twoprev.a - average.twoprev.a +
          vec->twonext.a - average.twonext.a, 2) +
     pow(vec->twoprev.g - average.twoprev.g +
          vec->twonext.g - average.twonext.g, 2) +
     pow(vec->twoprev.t - average.twoprev.t +
          vec->twonext.t - average.twonext.t, 2) +
*/     pow((vec->twonext.c - average.twonext.c), 2) +
     pow((vec->twonext.a - average.twonext.a), 2) +
     pow((vec->twonext.g - average.twonext.g), 2) +
     pow((vec->twonext.t - average.twonext.t), 2) //+
/*     pow((vec->threenext.c - average.threenext.c), 2) +
     pow((vec->threenext.a - average.threenext.a), 2) +
     pow((vec->threenext.g - average.threenext.g), 2) +
     pow((vec->threenext.t - average.threenext.t), 2) +
     pow(vec->threeprev.c - average.threeprev.c +
          vec->threenext.c - average.threenext.c, 2) +
     pow(vec->threeprev.a - average.threeprev.a +
          vec->threenext.a - average.threenext.a, 2) +
     pow(vec->threeprev.g - average.threeprev.g +
          vec->threenext.g - average.threenext.g, 2) +
     pow(vec->threeprev.t - average.threeprev.t +
          vec->threenext.t - average.threenext.t, 2)
*/     ) *
           (pow(0.3*(vec->timetocall - average.timetocall), 2) + 1) // *
//        (pow(0.01*(vmax/vlast - amax/alast), 2) + 1)
//        ((vmax < amax)? (pow(0.5*(vmax - amax), 2) + 1) : 1)
//        (pow((log(vec->twoprevmax) - log(average.twoprevmax))*.25, 2) +
//         pow((log(vec->twonextmax) - log(average.twonextmax))*.25, 2))
           );


//  if (vec->lastcall == average.lastcall) dist = dist-1000;

  return(dist);
}

int vector::call(cluster **clust) {
  /* determines tag of closest cluster */

  double dist[NUMCLUSTS];
  int i;

  for (i=0; i < NUMCLUSTS; i++) {
    dist[i] = clust[i]->distance(this);
//    clust[lastcall*4 + i]->distance(this);
  }

  /* note: for coding purposes, use linear min.  Should change
     to log time min by pairing */
  double min;

  /* go through array updating min as you go */
  min = dist[0];
  for (i = 1; i < NUMCLUSTS; i++) {
    if (dist[i] < min) min = dist[i];
  }

  /* min is now minimum distance - determine cluster */
//  if (min < 10000) {
  /* set currmin to this value */
  currmin = min;
//  printf("\n%15.0f ", min);
  for (i = 0; i < NUMCLUSTS; i++) {
    if (min == dist[i]) {
/*      printf("Current vector: \n");
      this->print();
      printf("Closest cluster: \n");
      clust[i]->average.print();    */
/*        printf(" (%d) ", i); */
      currcluster = i;
      return (clust[i]->average.tag);
    }
  }
//}

  /* default */
  return(-1);
}

int vector::findclust() {
  /* returns index of cluster to which this vector belongs */
  return(4*lastcall + curr.tag);
}

double avedist(int prev, int curr, cluster **clust) {
  /* find cluster and return average timetocall */
  return(clust[4*prev + curr]->average.timetocall);
}

int clusterprevtag(int cl) {
  /* return tag of previous base of cluster number cl */
  return (cl/4);
}

int clustercurrtag(int cl) {
  /* return tag of current base of cluster number cl */
  return (cl%4);
}

void averagetwovecs(vector *vec1, vector *vec2, vector *average) {
  averagetwodps(&vec1->threeprev, &vec2->threeprev, &average->threeprev);
  averagetwodps(&vec1->twoprev, &vec2->twoprev, &average->twoprev);
  averagetwodps(&vec1->prev, &vec2->prev, &average->prev);
  averagetwodps(&vec1->curr, &vec2->curr, &average->curr);
  averagetwodps(&vec1->next, &vec2->next, &average->next);
  averagetwodps(&vec1->twonext, &vec2->twonext, &average->twonext);
  averagetwodps(&vec1->threenext, &vec2->threenext, &average->threenext);
  if (vec1->lastcall == vec2->lastcall) average->lastcall = vec1->lastcall;
  else {
    vec1->lastcall = -1;
    fprintf(stderr, "Warning: averaging unmatched vectors.\n");
  }
  average->timetocall = (vec1->timetocall + vec2->timetocall)/2;
  average->lastcallval = (vec1->lastcallval + vec2->lastcallval)/2;
  average->max = (vec1->max + vec2->max)/2;
  average->twoprevmax = (vec1->twoprevmax + vec2->twoprevmax)/2;
  average->twonextmax = (vec1->twonextmax + vec2->twonextmax)/2;
  if (vec1->tag == vec2->tag) average->tag = vec1->tag;
  else {
    average->tag = -1;
    fprintf(stderr, "Warning: averaging unmatched vectors.\n");
  }
}


void averagetwodps(datapoint *dp1, datapoint *dp2, datapoint *average) {
  average->c = (dp1->c + dp2->c)/2;
  average->t = (dp1->t + dp2->t)/2;
  average->a = (dp1->a + dp2->a)/2;
  average->g = (dp1->g + dp2->g)/2;
  if (dp1->tag == dp2->tag) average->tag = dp1->tag;
  else {
    average->tag = -1;
    fprintf(stderr, "Warning: averaging unmatched datapoints %c %c.\n",
             inttocall(dp1->tag), inttocall(dp2->tag));
  }
}
```

```
extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
}

main(int argc, char **argv) {
  FILE *datafile, *seqfile;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <datafile> <sequence file>\n", argv[0]);
    exit(1);
  }

  datafile = fopen(argv[1], "r");
  seqfile = fopen(argv[2], "r");

  double c, a, g, t;
  char call[2], realcall[2];

  if (!datafile) {
    fprintf(stderr, "Can not open data file %s.\n", argv[1]);
    exit(1);
  }

  if (!seqfile) {
    fprintf(stderr, "Can not open sequence file %s.\n", argv[2]);
    exit(1);
  }

  int line = 0, base = 0;

  while ((fscanf(datafile, "%lf%lf%lf%lf%1s", &c, &a, &g, &t, call)) != 5) {
    line++;
    if (call[0] != 'X') {
      base++;
      if (fscanf(seqfile, "%1s", realcall) != 1) {
         fprintf(stderr, "Sequence file ends too early!\n");
         exit(1);
      }
      if (call[0] != realcall[0]) {
         fprintf(stderr, "Sequence discrepancy at line %d, base %d\n",
                  line, base);
         exit(1);
      }
    }
  }
  printf("Sequences match.\n");

  fclose(datafile);
  fclose(seqfile);

  exit(0);
}
```

```
extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
}

main(int argc, char *argv[]) {
  FILE *pfile, *rfile, *yfile, *gfile, *infile;

  char prefix[32], *infilename;
  int i=0;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <datafile>\n", argv[0]);
    exit(1);
  }

  infilename = argv[1];

  if (!(infile = fopen(infilename, "r"))) {
    fprintf(stderr, "%s: can not open file %s\n", argv[0], infilename);
    exit(1);
  }

  while ((infilename[i] != '.') && (infilename[i] != '\0')) {
    prefix[i] = infilename[i];
    i++;
  }
  prefix[i] = '.';
  char *prefixend;
  prefixend = &prefix[++i];

  sprintf(prefixend, "pur\0");
  printf("%s\n", prefix);
  pfile = fopen(prefix, "w");
  sprintf(prefixend, "red\0");
  printf("%s\n", prefix);
  rfile = fopen(prefix, "w");
  sprintf(prefixend, "yel\0");
  printf("%s\n", prefix);
  yfile = fopen(prefix, "w");
  sprintf(prefixend, "grn\0");
  printf("%s\n", prefix);
  gfile = fopen(prefix, "w");

  double p, r, y, g;
  while (fscanf(infile, "%lf %lf %lf %lf\n", &p, &r, &y, &g) == 4) {
    fprintf(pfile, "%.0f\n", p);
    fprintf(rfile, "%.0f\n", r);
    fprintf(yfile, "%.0f\n", y);
    fprintf(gfile, "%.0f\n", g);
  }

  fclose(pfile);
  fclose(rfile);
  fclose(yfile);
  fclose(gfile);
  fclose(infile);

  exit(0);
}
```

```
/* zips together files into standard output */

extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
}

main(int argc, char **argv) {
  FILE *pfile, *rfile, *yfile, *gfile;

  if (argc < 5) {
    fprintf(stderr,
              "Usage: %s <purple file> <red file> <yellow file> <green file>\n",
              argv[0]);
    exit(1);
  }

  pfile = fopen(argv[1], "r");
  rfile = fopen(argv[2], "r");
  yfile = fopen(argv[3], "r");
  gfile = fopen(argv[4], "r");

  double p, r, y, g;

  while ((fscanf(pfile, "%lf", &p) == 1) &&
         (fscanf(rfile, "%lf", &r) == 1) &&
         (fscanf(yfile, "%lf", &y) == 1) &&
         (fscanf(gfile, "%lf", &g) == 1)) {
    printf("%10.0f %10.0f %10.0f %10.0f\n", p, r, y, g);
  }

  fclose(pfile);
  fclose(rfile);
  fclose(yfile);
  fclose(gfile);

  exit(0);
}
```

```
/* Background subtraction.  Takes four minima: current - BIGWIN to
   current, current - SMALLWIN to current, current to current +
   SMALLWIN, and current to current + BIGWIN.  Fits a line to these points
   using code from p. 527 of Numerical Recipes in C.  Evaluates this line
   at the current point and subtracts that value.  Writes to standard output.
   If argument, reads input from argument file, otherwise reads from standard
   input.
*/

extern "C" {
  #include <stdio.h>
  #include <stdlib.h>
  #include <math.h>
}

#include <linefit.h>

#define SMALLWIN 50
#define BIGWIN   100

class entry {
 public:
  double val;
  int time;
  entry *prevval;
  entry *nextval;
  entry *prevtime;
  entry *nexttime;
  entry(double v = 0, int t = 0) {val = v; time = t;};
  void init(double v = 0, int t = 0) {val = v; time = t;};
};

class window {
  entry *lowval = NULL;
  entry *lowtime = NULL;
  entry *hightime = NULL;
 public:
  void insert(entry *);
  entry *clearlowtime();
  void walkvals();
  void walktime();
  double min(){return lowval->val;};
  int mintime(){return lowval->time;};
};

class channel {
 public:
  window smallprevwin, bigprevwin, smallnextwin, bignextwin;
  double bigprevmin, smallprevmin, smallnextmin, bignextmin;
  int bigprevmintime, smallprevmintime, smallnextmintime, bignextmintime;
  int previndex, smallnextindex, bignextindex;
  int prevtime, smallnexttime, bignexttime;

  double buf[2*BIGWIN - 1];
  int bufsize = 0;
  void walkalltimes();
  void walkallvals();
  void setallmins();
  void printallmins();
  void bufinit(FILE *);
  void timezero();
  void firstphase();
  void secondphase();
  void mainphase(FILE *);
  void endphase();
  double fitcurrtime();
  void subtract();
};

void channel::subtract() {
 printf("%.0f\n", buf[previndex] - fitcurrtime());
//  printf("%.0f\n", fitcurrtime());  /* To get background */
}

double channel::fitcurrtime() {
//  walkalltimes();
  setallmins();
//  printallmins();
//  printf("Fitting line: ");

  /* if all are equal, return that value */
  if ((bigprevmintime == bignextmintime) &&
      (smallprevmintime == smallnextmintime) &&
      (bigprevmintime == smallprevmintime))
    {
//      printf("all points are equal\n");
//      printf("Subtraction value at %d is %.2f\n",
//             prevtime, bigprevmin);
      return (bigprevmin);
    }

  /* otherwise fit to a line */
  double x[4] = {double (bigprevmintime), double (smallprevmintime),
                     double (smallnextmintime), double (bignextmintime)};
  double y[4] = {bigprevmin, smallprevmin, smallnextmin, bignextmin};
  double *xx = x-1;
  double *yy = y-1;
  double sig[4];
  double a, b, siga, sigb, chi2, q;

  fit(xx, yy, 4, sig, 0, &a, &b, &siga, &sigb, &chi2, &q);

//  printf("y = %.2f + %.2f(x)\n", a, b);
//  printf("Subtraction value at %d is %.2f\n",
//        prevtime, a + b*(double(prevtime)));
   return (a + b*(double(prevtime)));
}

void channel::endphase() {
  /* scroll off ends */
  int i;
  entry *curr;

  for (i = 0; i < SMALLWIN; i++) {
    /* scroll all but bignextwin, drop low entry for bignextwin */
    curr = bigprevwin.clearlowtime();
    curr->init(buf[previndex], prevtime);
    bigprevwin.insert(curr);

    curr = smallprevwin.clearlowtime();
    curr->init(buf[previndex], prevtime);
    smallprevwin.insert(curr);

    curr = smallnextwin.clearlowtime();
    curr->init(buf[smallnextindex], smallnexttime);
    smallnextwin.insert(curr);

    curr = bignextwin.clearlowtime();
    delete curr;

    subtract();

    prevtime++;
    smallnexttime++;
    previndex = (previndex+1)%bufsize;
    smallnextindex = (smallnextindex+1)%bufsize;

   }
  for (i = SMALLWIN; i < BIGWIN - 1; i++) {
    /* scroll previous windows, drop low entry for nextwindows */
    curr = bigprevwin.clearlowtime();
    curr->init(buf[previndex], prevtime);
    bigprevwin.insert(curr);

    curr = smallprevwin.clearlowtime();
    curr->init(buf[previndex], prevtime);
    smallprevwin.insert(curr);

    curr = smallnextwin.clearlowtime();
    delete curr;
    curr = bignextwin.clearlowtime();
    delete curr;

    subtract();

    prevtime++;
    previndex = (previndex+1)%bufsize;
  }
}

void channel::mainphase(FILE *infile) {
  int i;
  entry *curr;

  /* all windows are full - continue moving current time forward */
  while ((fscanf(infile, "%lf", &buf[bignextindex]) == 1)) {
    /* scroll all windows */
    curr = bigprevwin.clearlowtime();
    curr->init(buf[previndex], prevtime);
    bigprevwin.insert(curr);

    curr = smallprevwin.clearlowtime();
    curr->init(buf[previndex], prevtime);
    smallprevwin.insert(curr);

    curr = smallnextwin.clearlowtime();
    curr->init(buf[smallnextindex], smallnexttime);
    smallnextwin.insert(curr);

    curr = bignextwin.clearlowtime();
    curr->init(buf[bignextindex], bignexttime);
    bignextwin.insert(curr);
    subtract();

    previndex = (previndex+1)%bufsize;
    smallnextindex = (smallnextindex+1)%bufsize;
    bignextindex = (bignextindex+1)%bufsize;
    prevtime++;
    smallnexttime++;
    bignexttime++;
  }
}

void channel::secondphase() {
  int i;
  entry *curr;

  /* bigprevwin is still not full - continue moving current time forward */
  for (i = BIGWIN + SMALLWIN - 1; i < 2*BIGWIN - 1; i++) {
    /* insert into bigprev window, scroll other windows */
    curr = new entry(buf[previndex], prevtime);
    bigprevwin.insert(curr);

    curr = smallprevwin.clearlowtime();
    curr->init(buf[previndex], prevtime);
    smallprevwin.insert(curr);

    curr = smallnextwin.clearlowtime();
    curr->init(buf[smallnextindex], smallnexttime);
    smallnextwin.insert(curr);

    curr = bignextwin.clearlowtime();
    curr->init(buf[bignextindex], bignexttime);
    bignextwin.insert(curr);

    subtract();

    previndex = (previndex+1)%bufsize;
    smallnextindex = (smallnextindex+1)%bufsize;
    bignextindex = (bignextindex+1)%bufsize;
    prevtime++;
    smallnexttime++;
    bignexttime++;
  }
}

void channel::firstphase() {
  /* move current time forward - still have incomplete prev windows */
  int i;
  entry *curr;

  for (i = BIGWIN; i < BIGWIN + SMALLWIN - 1; i++) {
    /* insert into previous windows, scroll next windows */
    curr = new entry(buf[previndex], prevtime);
    bigprevwin.insert(curr);

    curr = new entry(buf[previndex], prevtime);
    smallprevwin.insert(curr);

    curr = smallnextwin.clearlowtime();
    curr->init(buf[smallnextindex], smallnexttime);
    smallnextwin.insert(curr);

    curr = bignextwin.clearlowtime();
    curr->init(buf[bignextindex], bignexttime);
    bignextwin.insert(curr);

    subtract();

    previndex = (previndex+1)%bufsize;
    smallnextindex = (smallnextindex+1)%bufsize;
    bignextindex = (bignextindex+1)%bufsize;
    prevtime++;
    smallnexttime++;
    bignexttime++;
  }
}

void channel::timezero() {
  entry *curr;

  /* initialize to current time 0 */
  previndex = prevtime = 0;
  smallnextindex = smallnexttime = 0;
  bignextindex = bignexttime = 0;

  /* insert entry 0 into all windows */
  curr = new entry(buf[previndex], previndex);
  bigprevwin.insert(curr);

  curr = new entry(buf[previndex], previndex);
  smallprevwin.insert(curr);

  curr = new entry(buf[smallnextindex], smallnextindex);
  smallnextwin.insert(curr);

  curr = new entry(buf[bignextindex], bignextindex);
  bignextwin.insert(curr);

  previndex = (previndex+1)%bufsize;
  smallnextindex = (smallnextindex+1)%bufsize;
  bignextindex = (bignextindex+1)%bufsize;
  prevtime++;
  smallnexttime++;
  bignexttime++;

  int i;

  for (i = 1; i < SMALLWIN; i++) {
    /* insert into smallnextwin and bignextwin*/
    curr = new entry(buf[smallnextindex], smallnextindex);
    smallnextwin.insert(curr);

    curr = new entry(buf[bignextindex], bignextindex);
    bignextwin.insert(curr);

    smallnextindex = (smallnextindex+1)%bufsize;
    bignextindex = (bignextindex+1)%bufsize;
    smallnexttime++;
    bignexttime++;
  }
  for (i = SMALLWIN; i < BIGWIN; i++) {
    /* insert into bignextwin */
    curr = new entry(buf[bignextindex], bignextindex);
    bignextwin.insert(curr);

    bignextindex = (bignextindex+1)%bufsize;
    bignexttime++;
  }

  subtract();
}

void channel::bufinit(FILE *infile) {
  /* read in buffer */
  while ((bufsize < 2*BIGWIN - 1) &&
          (fscanf(infile, "%lf", &buf[bufsize]) == 1))
    bufsize++;

  if (bufsize < 2*BIGWIN - 1) {
    fprintf(stderr, "Input too short for window size %d\n", BIGWIN);
    exit(1);
  }
}

void window::insert(entry *e) {
  entry *p;

  /* insert in value list */
  p = lowval;

  if (p == NULL) { /* list is empty - make first element */
    e->prevval = NULL;
    e->nextval = NULL;
    lowval = e;
  }
  else {
    while ((p->val < e->val) && (p->nextval != NULL)) p = p->nextval;
    if (e->val <= p->val) { /* insert before p */
      e->prevval = p->prevval;
      p->prevval = e;
      e->nextval = p;
      if (e->prevval != NULL) e->prevval->nextval = e;
      if (e->val <= lowval->val) lowval = e;
    }
    else { /* reached end of list - insert at end */
      e->prevval = p;
      p->nextval = e;
      e->nextval = NULL;
    }
  }

  /* insert in time list */
  if (lowtime == NULL) {  /* list is empty - make first element */
    e->prevtime = NULL;
    e->nexttime = NULL;
    lowtime = e;
    hightime = e;
  }
  else {
    e->prevtime = hightime;
    if (e->prevtime != NULL) e->prevtime->nexttime = e;
    e->nexttime = NULL;
    hightime = e;
  }
};
```

```
entry *window::clearlowtime() {
  /* removes low time entry from the list, returns pointer
     to entry for reuse */
  entry *temp;

  temp = lowtime;
  if (lowtime == NULL) return NULL;
  if (lowval == lowtime) lowval = lowval->nextval;
  if (hightime == lowtime) hightime = NULL;
  lowtime = lowtime->nexttime;
  if (lowtime != NULL) lowtime->prevtime = NULL;
  if (temp->nextval != NULL) temp->nextval->prevval = temp->prevval;
  if (temp->prevval != NULL) temp->prevval->nextval = temp->nextval;

  return(temp);
};

void window::walkvals() {
  entry *temp = lowval;

  while (temp != NULL) {
    printf("%.0f ", temp->val);
    temp = temp->nextval;
  }
  printf("\n");
};

void window::walktime() {
  entry *temp = lowtime;

  while (temp != NULL) {
    printf("%.0f ", temp->val);
    temp = temp->nexttime;
  }
  printf("\n");
};

void channel::walkalltimes() {
  bigprevwin.walktime();
  smallprevwin.walktime();
  smallnextwin.walktime();
  bignextwin.walktime();
  printf("\n");
}
```

```
void channel::walkallvals() {
  bigprevwin.walkvals();
  smallprevwin.walkvals();
  smallnextwin.walkvals();
  bignextwin.walkvals();
  printf("\n");
}

void channel::setallmins() {
  bigprevmin = bigprevwin.min();
  smallprevmin = smallprevwin.min();
  smallnextmin = smallnextwin.min();
  bignextmin = bignextwin.min();
  bigprevmintime = bigprevwin.mintime();
  smallprevmintime = smallprevwin.mintime();
  smallnextmintime = smallnextwin.mintime();
  bignextmintime = bignextwin.mintime();
}

void channel::printallmins() {
  printf("\nMins: (%d, %.0f) (%d, %.0f) (%d, %.0f) (%d, %.0f)\n",
          bigprevmintime, bigprevmin, smallprevmintime, smallprevmin,
          smallnextmintime, smallnextmin, bignextmintime, bignextmin);
}

main(int argc, char *argv[]) {
  FILE *infile;

//  malloc_debug(8);

  /* read from standard input if no arguments, otherwise read from
     argument file */
  if (argc < 2) infile = stdin;
  else if (!(infile = fopen(argv[1], "r"))) {
    fprintf(stderr, "%s: can not open file %s\n", argv[0], argv[1]);
    exit(1);
  }

  entry *curr;
  channel c;

  c.bufinit(infile);
  c.timezero();
  c.firstphase();
  c.secondphase();
  c.mainphase(infile);
```

```
    c.endphase();

    fclose(infile);

    exit(0);
  }
```

What is claim d is:

1. An integrated apparatus for concurrent preparation and analysis of a plurality of biopolymer fragment samples, each sample comprising a plurality of fragments obtained from one or more biopolymers, the apparatus comprising:

(a) means for substantially concurrent electrophoretic separation of each of a plurality of biopolymer fragment samples loaded into an electrophoretic separation medium;

(b) means for substantially simultaneously stimulating light emissions from fragments in a plurality of biopolymer fragment samples; and

(c) means for substantially simultaneous resolution of said light emissions into spatial and spectral components and generation of output signals representative thereof.

2. The apparatus of claim 1 further comprising means for the analysis of the detected light emission to give information on identity of the biopolymer samples.

3. The apparatus of claim 1 further comprising means for loading the plurality of biopolymer fragment samples into the electrophoretic separation medium.

4. The apparatus of claim 3 wherein the means for loading the plurality of biopolymer fragment samples comprises:

(a) a plurality of wells in the electrophoretic separation medium from which the biopolymer fragment samples migrate for separation, each well containing a buffer medium;

(b) a solid phase loading comb having a plurality of teeth, each tooth being spaced and sized to fit into one of the plurality of wells, the teeth having means for the adhesion of the biopolymer fragment samples, the adhesion being such that the fragment samples are released upon insertion into the buffer medium in the wells.

5. The apparatus of claim 4 wherein the biopolymer fragment samples are DNA sequencing reaction fragment samples, the means for the adhesion of the samples are a plurality of sequencing templates bound to the teeth of the comb, the release upon insertion into the wells occurring with denaturation of the bound fragments from the templates.

6. The apparatus of claim 4 in which the solid phase loading comb is guided into the sample wells by notches formed on a plate, the notches being sized to match the teeth of the comb.

7. The apparatus of claim 3 wherein the means for loading the plurality of biopolymer fragment samples comprises:

(a) a plurality of wells in the electrophoretic separation medium from which the biopolymer fragment samples migrate for separation, each well containing a buffer medium; and

(b) a solid phase loading system comprising a plurality of magnetic beads, such beads being placed into the plurality of wells, the beads having means for adhesion of the biopolymer fragment samples, the adhesion being such that the fragment samples are released upon insertion into the buffer medium in the wells.

8. The apparatus of claim 3 wherein the means for loading further comprises a sample focusing electrode and a means for controlling voltage applied to said electrode for the purpose of concentrating loaded samples.

9. The apparatus of claim 1 further comprising means for preparing from an input sample of a plurality of biopolymers a plurality of biopolymer fragment samples for subsequent analysis.

10. The apparatus of claim 1 wherein the means for substantially concurrent electrophoretic separation is a electrophoretic module comprising:

(a) a substantially flat bottom plate; and

(b) a substantially flat top plate, the top plate being positioned above the bottom plate for forming a narrow cavity to hold the electrophoretic separation medium.

11. The apparatus of claim 10 wherein the top and bottom plates are separated by approximately 25 μm to approximately 250 μm.

12. The apparatus of claim 10 further comprising thermal control means for maintaining a selected uniform temperature in the bottom plate.

13. The apparatus of claim 12 wherein the thermal control means further comprises:

(a) a heat sink for exchanging heat with the surroundings; and

(b) a plurality of thermal transfer devices disposed between and in thermal contact with the heat sink and the bottom plate for bi-directional heat transfer.

14. The apparatus of claim 13 wherein the thermal transfer devices are Peltier thermo-electric devices.

15. The apparatus of claim 10 wherein electrophoretic migration lanes are formed between the top and bottom plates.

16. The apparatus of claim 1 wherein the means for substantially concurrent electrophoretic separation is an electrophoresis module comprising:

(a) a substantially flat bottom plate; and

(b) a substantially flat top plate having a plurality of grooves on one side, the top plate being positioned in contact with the bottom plate so that the grooves form with the bottom plate a plurality of separated channels, the channels holding the electrophoretic medium to form electrophoretic migration lanes.

17. The apparatus of claim 16 wherein the grooves in the top plate are from approximately 25 μm to approximately 250 μm in cross-section.

18. The apparatus of claim 16 wherein the grooves in the top plate are straight and spaced to be parallel.

19. The apparatus of claim 16 wherein the grooves in the top plate are spaced to converge toward one end of the plate.

20. The apparatus of claim 16 wherein the top plate is glass and the lanes are formed by etching.

21. The apparatus of claim 20 wherein the top plate is borofloat glass and the lanes are formed by hydrogen fluoride etching.

22. The apparatus of claim 16 wherein the grooved top plate is cast in a mold.

23. The apparatus of claim 16 wherein the top plate is disposable.

24. The apparatus of claim 16 in which the grooves are formed in polymer.

25. The apparatus of claim 16 further comprising:

(a) cross-lane grooves between selected adjacent lanes forming cross-lane connecting channels, the channels holding separation medium so that biopolymer fragments can migrate between the selected adjacent lanes through the channels; and

(b) electrodes formed in the walls of the selected lanes adjacent to the cross-lane connecting channels for causing fragment migration through the connecting channels upon being energized with voltage.

26. The apparatus of claim 16 in which either the top plate or the bottom plate is a thermal conductor coated with an electrical insulator, and the grooves are formed on the insulator.

27. The apparatus of claim 1 in which the separation medium comprises small posts fabricated directly in the migration lanes.

28. The apparatus of claim 1 in which the separation medium comprises small spheres of an inert material such as polystyrene.

29. The apparatus of claim 1 wherein the means for substantially simultaneous resolution of each of a plurality of light emissions is a transmission imaging spectrograph comprising:

(a) an optic assembly positioned to receive a substantial fraction of the plurality of light emissions for simultaneous spatial imaging along a first axis and spectral dispersion along a second axis; and

(b) a detector array for simultaneous spatial and spectral detection of the plurality of light emissions imaged along the first axis and dispersed along the second axis by the optic assembly, the detector producing output signals representative of the detected light.

30. The apparatus of claim 29 wherein the detector array is a CCD array producing electronic output signals.

31. The apparatus of claim 29 wherein the optic assembly further comprises binary optics for spectrally dispersing the light emissions along the first axis and for focusing the

light emissions along the second axis onto the detector array.

32. The apparatus of claim 29 wherein the optic assembly further comprises:

(a) a collection lens positioned to initially receive and collimate the plurality of light emissions;

(b) a transmission dispersion element for spectrally dispersing the collimated signals along the first axis; and

(c) a focusing lens for spatially focusing the signals along the second axis onto the detector array.

33. The apparatus of claim 32 further comprising a spectral filter element positioned between the collection lens and transmission dispersion element for filtering from the light emissions extraneous light.

34. The apparatus of claim 32 in which the transmission dispersion element is a transmission diffraction grating, or a transmission grating-prism.

35. The apparatus of claim 1 wherein the biopolymer fragment samples are labeled with spectrally distinctive dyes and further comprising a laser illuminating the separated biopolymer fragment samples to stimulate light emission from the dye labels.

36. The apparatus of claim 35 wherein the fragment samples are illuminated at multiple wavelengths.

37. The apparatus of claim 35 wherein the laser is a solid state laser.

38. The apparatus of claim 3 wherein the means for loading the plurality of biopolymer fragment samples for separation comprises:

(a) an array of micro-reactors in which biopolymer fragment samples are generated from biopolymer samples, each micro-reactor having a minute inlet for loading a biopolymer sample;

(b) a plurality of capillary inlet passages, each micro-reactor having an inlet passage, through which reagents needed for fragment generation are loaded;

(c) a plurality of capillary outlet passages, each micro-reactor having an outlet passage, through which biopolymer fragment samples are ejected into the electrophoresis module for separation; and

(d) a plurality of capillary controllers, one controller in each capillary inlet and outlet passage, for controlling fluid flow in the capillaries.

39. The apparatus of claim 38 wherein the capillary controller for controlling fluid flow in a capillary comprises:

(a) an electrical micro-heating element in thermal contact with the capillary; and

(b) electrical leads to the heating elements for energizing the heating elements, whereby current in the leads to a micro-heating element causes fluid evaporation in the contacted capillary and formation of a vapor bubble which blocks fluid flow in the capillary.

40. The apparatus of claim 39 wherein the micro-heating elements comprise:

(a) a layer of a resistive material deposited adjacent to the contacted capillary; and

(b) a layer of protective material deposited over the layer of resistive material for separating the resistive layer from the capillary contents.

41. The apparatus of claim 1 wherein the biopolymer samples are DNA samples, the biopolymer fragment samples are DNA sequencing reaction fragments labeled with dyes, each dye

having distinctive spectral properties, said apparatus further comprising:

(a) memory means for storing the output signals of the means for resolution and detection and for storing a set of prototype signals;

(b) means for cumulating the stored output signals into spectral samples, each such spectral sample representative of the distinctive spectral characteristics of the dye labels of one biopolymer fragment sample,

(c) means for comparing the time behavior of the spectral samples with the time behaviors of the set of prototype signals and for selecting prototypes from the set that most closely match the spectral samples; and

(d) means for outputting identities of the selected prototypes as the DNA sequences of the DNA samples.

42. The apparatus of claim 41 wherein the set of prototype signals comprises the output from the analysis of well known DNA sequences.

43. The apparatus of claim 41 wherein the selected prototypes represent pairs or triples of sequential DNA bases.

44. The apparatus of claim 41 wherein the means for selecting comprises comparing a distance metric between the time behaviors of the spectral samples and the time behaviors of the prototypes and selecting as representative that prototype with the closest distance.

45. The apparatus of claim 44 wherein the distance metric is output as an indication of the probable accuracy of the DNA sequences.

46. The apparatus of claim 44 wherein the distance metric is the sum of the squares of the differences in signal values between the spectral samples and the prototype signals.

47. The apparatus of claim 41 further comprising means for outputting identiti s of multiple samples and relationships between said sample identities.

48. The apparatus of claim 41 further comprising:

(a) means for trimming from the output DNA sequences of the DNA samples stored in memory known DNA sequences in the DNA sample;

(b) means for proofreading in a Monte Carlo manner the trimmed DNA sequences stored in memory, the means for proofreading comprising means for repetitively making at a random point in the trimmed output a random sequence alteration and evaluating sequence improvement until no further substantial sequence improvement occurs; and

(c) means for storing and outputting the improved sequences.

49. The apparatus of claim 48 wherein sequence improvement is evaluated by evaluating a probabilistic Boltzman condition on the difference in two distance metrics, one distance metric being between the original DNA sequence and the spectral samples, the other distance metric being between alternative DNA sequences and the spectral samples.

50. The apparatus of claim 1 wherein the biopolymer samples are DNA samples, the biopolymer fragment samples are DNA sequencing reaction fragments labeled with dyes, each dye having distinctive spectral properties, said apparatus further comprising:

(a) memory means for storing the output signals of the means for resolution and detection and for storing a set of prototype signals;

(b) means for cumulating the stored output signals into a time series of spectral samples, each such spectral sample representative of the distinctive spectral characteristics of the dye labels of one biopolymer fragment sample;

(c) means for comparing at a plurality of successive observation times the time behavior of the time series of spectral samples with the time behaviors of the set of prototype signals and for selecting a prototype from the set that most closely matches the spectral samples; and

(d) means for outputting the identity of the prototype that is the closest match.

51. A method for generating DNA sequence reaction fragments in one reaction chamber without an intermediate separation step the method comprising the sequential steps of:

(a) performing the polymerase chain reaction amplification step with dUTP rich PCR primers;

(b) fragmenting the dUTP primers with Uracil DNA Glycosylase into fragments ineffective as DNA polymerase primers; and

(c) performing the Sanger sequencing reactions.

52. The method of claim 51 wherein the dUTP rich PCR primers have dUTP residues spaced no more than approximately six bases apart.

53. The method of claim 51 performed in an array of micro-reactors for ejection onto a biopolymer separation apparatus.

54. A method for determining the DNA sequences of a plurality of DNA samples, the method using spectral signals obtained by spectrographic observation of electrophoretically separated labeled DNA fragments, the fragments being produced by the Sanger sequencing reactions and being labeled with dyes having distinctive spectral properties, the method comprising the sequential steps of:

(a) cumulating the spectrographic signals into a plurality of spatial samples, each spatial sample being representative of fragments of one DNA sample, and for each spatial sample, cumulating the spectrographic output signals into spectral samples, each spectral sample being representative of the distinctive spectral characteristics of one of the dye labels;

(b) comparing the time behavior of the spectral samples with a set of prototype signal time behaviors and selecting prototypes from the set that most closely match the spectral samples; and

(c) outputting the identities of the selected prototypes.

55. The method of claim 54 wherein the time behaviors of the set of prototype signals comprises the grouped output from the analysis of well known DNA sequences.

56. The method of claim 55 wherein the selected prototypes are pairs of sequential DNA fragments.

57. The method of claim 54 wherein the step of selecting comprises comparing a distance metric between the time behaviors of the spectral samples and the time behaviors of the prototypes and selecting as representative that prototype with the closest distance.

58. The method of claim 57 wherein the distance metric is the sum of the squares of the differences in signal values between the spectral samples and the prototypes.

59. The method of claim 57 wherein the distance metric is the product of (x), the sum of the squares of the differences in signal values between the spectral samples and the prototypes, and (y) a metric distance representing the difference between the prototypically expected time between observation times of closest match, and the actual time between observation times of closest match.

60. The method of claim 59 in which the metric distance (y) is a function of the square of the difference between the prototypically expected time between observation times of

closest match and the actual time between observation times of closest match.

61. The apparatus of claim 57 wherein the distance metric is output as an indication of the probable accuracy of the DNA sequences.

62. The method of claim 54 further comprising the steps of:

(a) trimming from the output identities of the DNA samples known DNA sequences in the DNA sample; and

(b) proofreading in a Monte Carlo manner the trimmed DNA sequences stored in memory, the step of proofreading comprising repetitively making at a random point in the trimmed output a random sequence alteration and evaluating sequence improvement until no further substantial sequence improvement occurs; and

(c) storing and outputting the improved sequence.

63. The method of claim 62 wherein evaluating sequence improvement is done by evaluating a probabilistic Boltzman condition on the difference in two distance metrics, one distance metric being between the original DNA sequence and the spectral samples, the other distance metric being between alternative DNA sequences and the spectral samples.

64. A method for determining the DNA sequences of a plurality of DNA samples, the method using spectral signals obtained by spectrographic observation of electrophoretically separated labeled DNA fragments, the fragments being produced by the Sanger sequencing reactions and being labeled with dyes having distinctive spectral properties, the method comprising the sequential steps of:

(a) storing the output signals of the means for resolution and detection and for storing a set of prototype signals;

(b) cumulating the stored output signals into a time series of spectral samples, each such spectral sample representative of the distinctive spectral characteristics of the dye labels of one biopolymer fragment sample;

(c) comparing at a plurality of successive observation times the time behavior of the time series of spectral samples with the time behaviors of the set of prototype signals and selecting a prototype from the set that most closely matches the spectral samples; and

(d) outputting the identity of the prototype that is the closest match.

65. An apparatus for the concurrent analysis of a plurality of DNA sequencing reaction fragment samples, each sample comprising a plurality of labeled DNA sequencing reaction fragments generated from one DNA sample, the labels being dyes with distinctive spectral characteristics, the apparatus comprising:

(a) means for loading on the apparatus the plurality of DNA fragment samples for separation;

(b) an electrophoresis module for the substantially concurrent separation of each of the plurality of DNA fragment samples, the electrophoresis module further comprising:

i. a substantially flat bottom plate; and

ii. a substantially flat top plate having a plurality of grooves on one side, the top plate being positioned in contact with the bottom plate so that the grooves form with the bottom plate a plurality of separated channels, the channels holding the electrophoretic medium to form electrophoretic migration lanes.

(c) a heat control subunit for maintaining a controlled uniform temperature in the electrophoresis module, the subunit further comprising:

i. a heat sink for exchanging heat with the surroundings;

ii. a plurality of thermal transfer devices disposed between and in thermal contact with the heat sink and the bottom plate for bi-directional heat transfer;

(d) a laser illuminating the separated biopolymer fragment samples to stimulate emission of the dye labels and thereby to generate the plurality of light signals;

(e) a transmission imaging spectrograph for substantially simultaneous detection of each of the plurality of light signals, the spectrograph further comprising:

i. an optic assembly positioned to receive a substantial fraction of the plurality of light signals for simultaneous spatial imaging along a first axis and spectral dispersion along a second axis; and

ii. a detector array for simultaneous spatial and spectral detection of the plurality of light signals imaged along the first axis and dispersed along the second axis by the optic assembly, the detector producing output signals representative of the detected light;

(f) means for the analysis of the detection output signals to determine the sequences of the DNA samples from the detector comprising:

i. memory means for storing output signal data, processed signal data, and prototype signal data;

ii. means for cumulating the detection output signals into spatial samples representative of one migration lane, and for each spatial sample, for cumulating the detection output signals into spectral samples representative of the distinctive spectral characteristics of the dye labels;

iii. means for comparing the time behavior of the spectral samples with a set of prototype time behaviors also stored in memory and for selecting prototypes that most closely match the spectral samples; and

iv. means for outputting identities of the selected prototypes as the DNA sequences of the DNA samples.

66. The apparatus of claim 65 wherein the apparatus further comprises:

(a) means for trimming from the output identities of the DNA samples known DNA sequences in the DNA sample;

(b) means for proofreading in a Monte Carlo manner the trimmed DNA sequences, the means for proofreading comprising repetitively making at a random point in the trimmed output a random sequence alt ration and evaluating sequence improvement until no further substantial sequence improvement occurs; and

(c) means for outputting the improved sequence as the DNA sequences of the DNA samples.

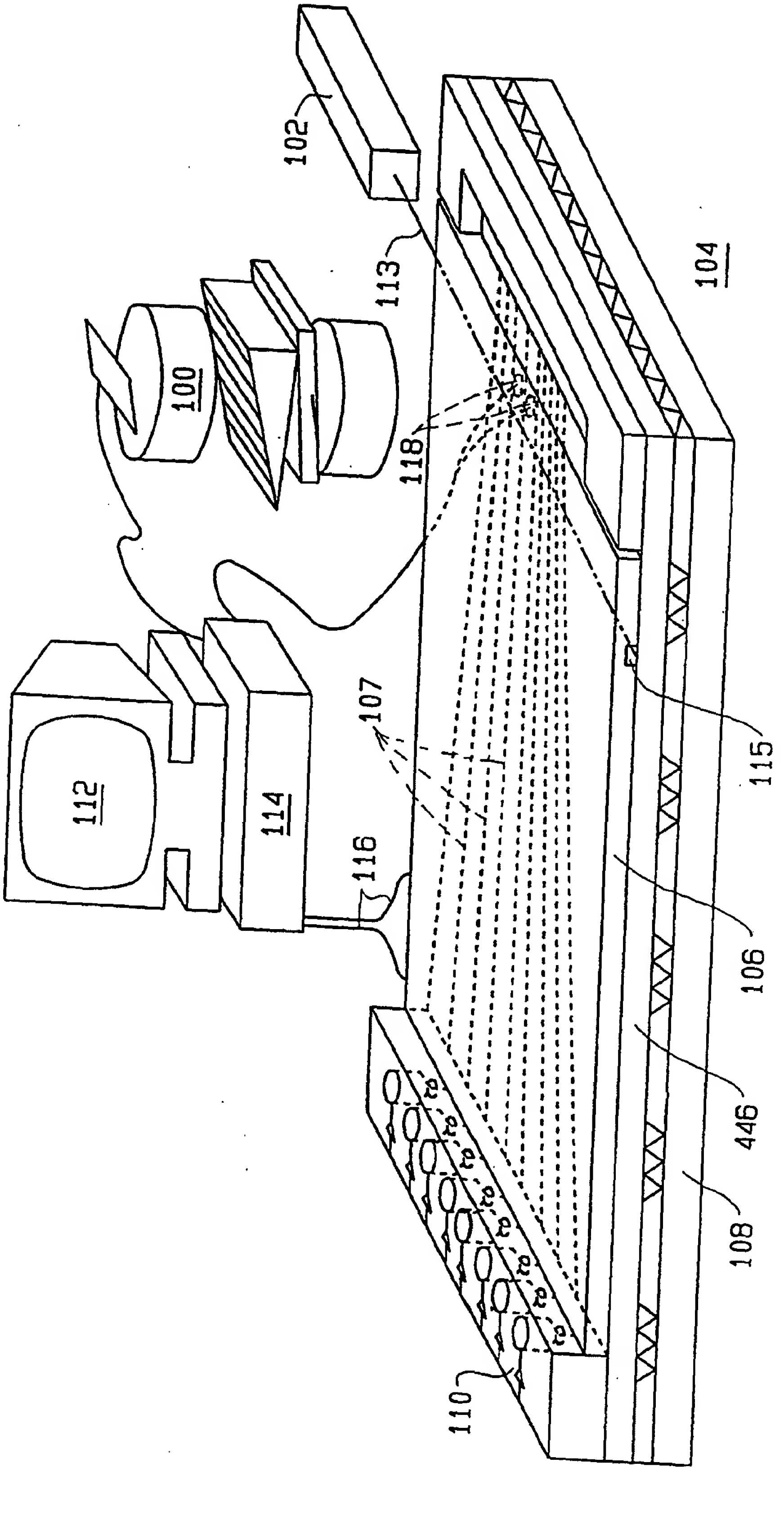


FIG. 1

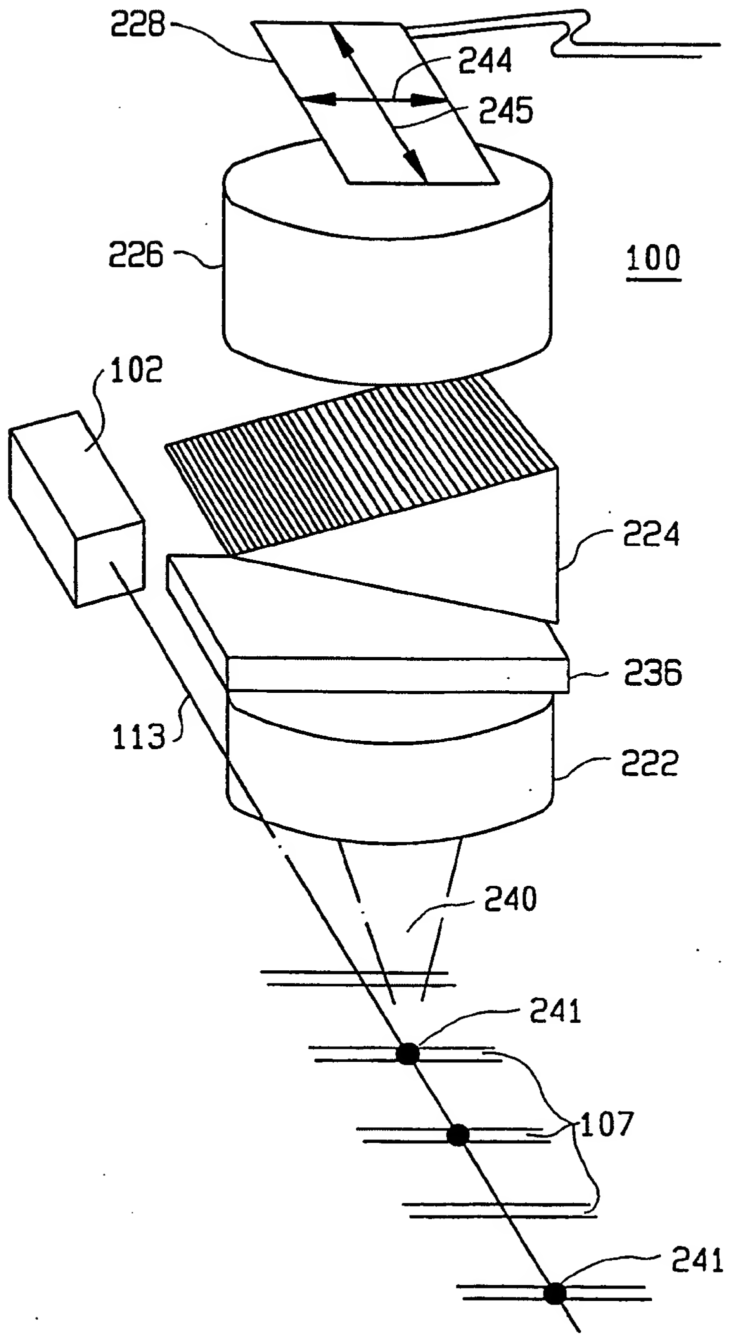


FIG. 2A

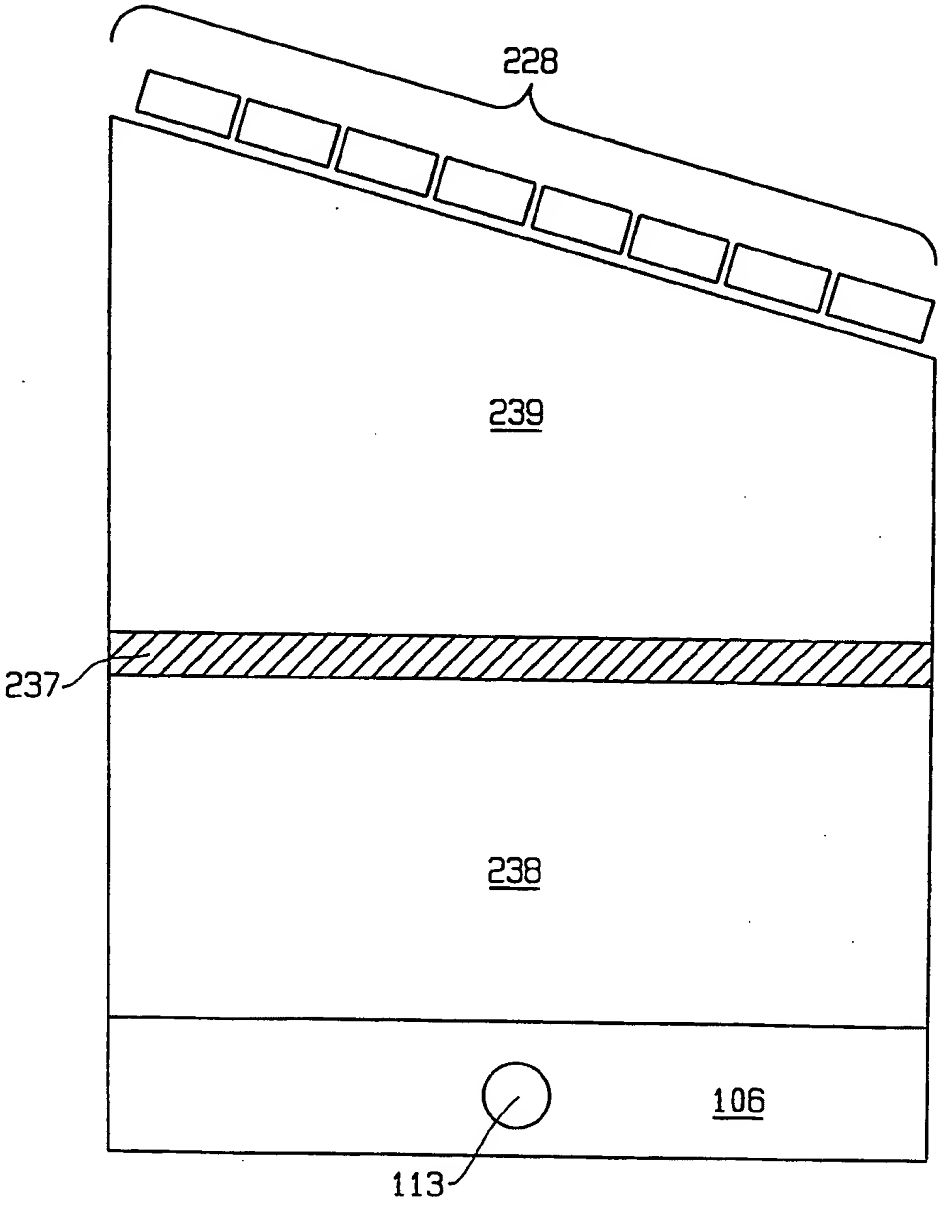


FIG. 2B

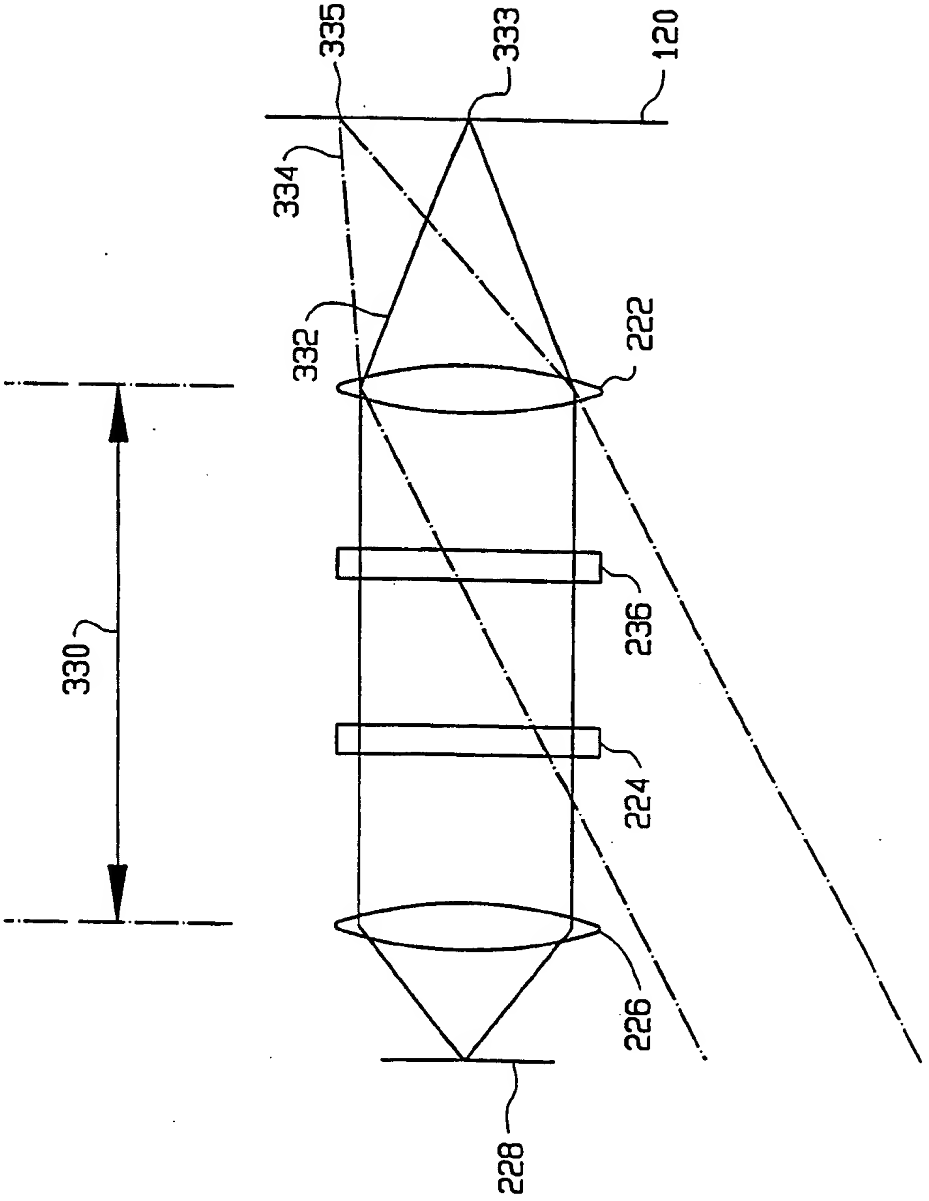


FIG. 3

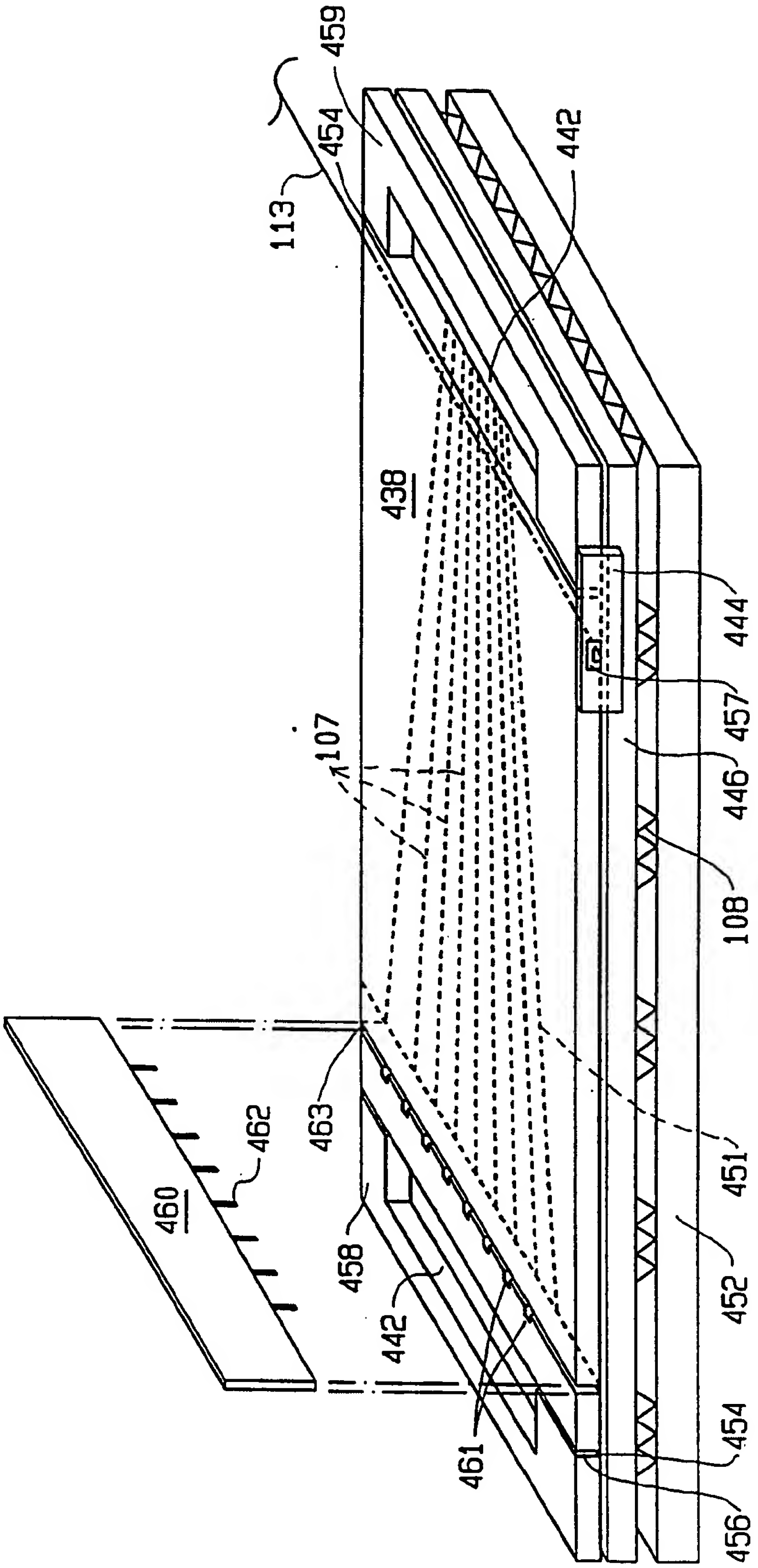

459
113
454
442
438
444
457
446
108
107
451
452
454
456
461
442
458
460
462
463

FIG. 4

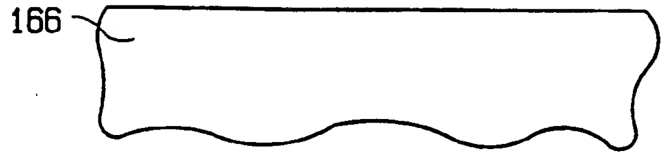


FIG. 5A

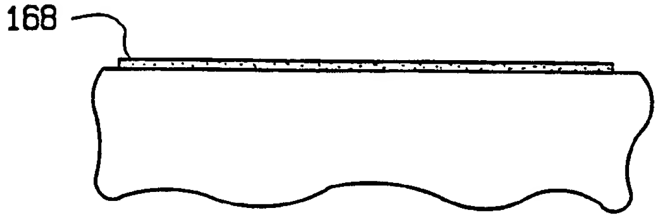


FIG. 5B

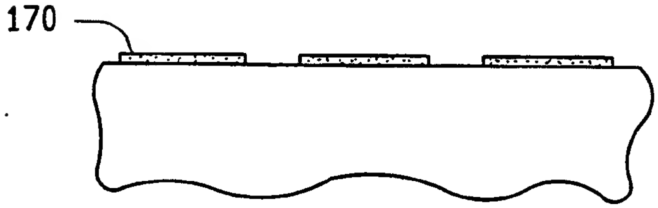


FIG. 5C

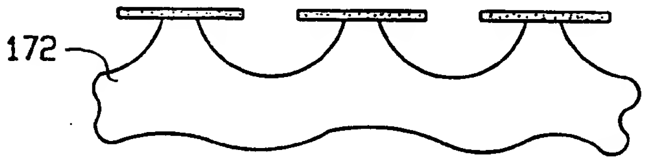


FIG. 5D

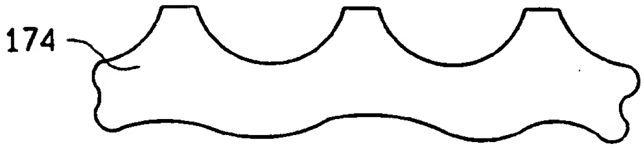


FIG. 5E

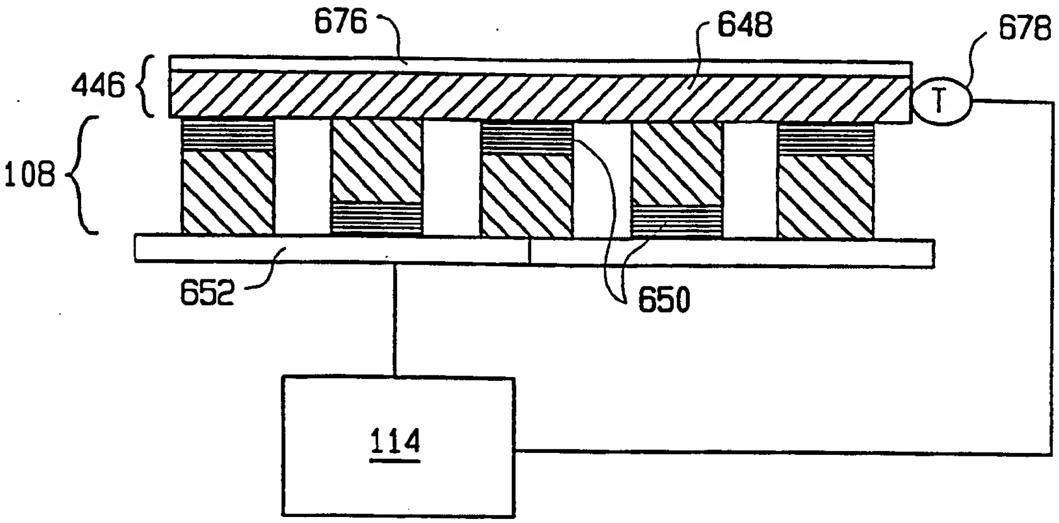


FIG. 6

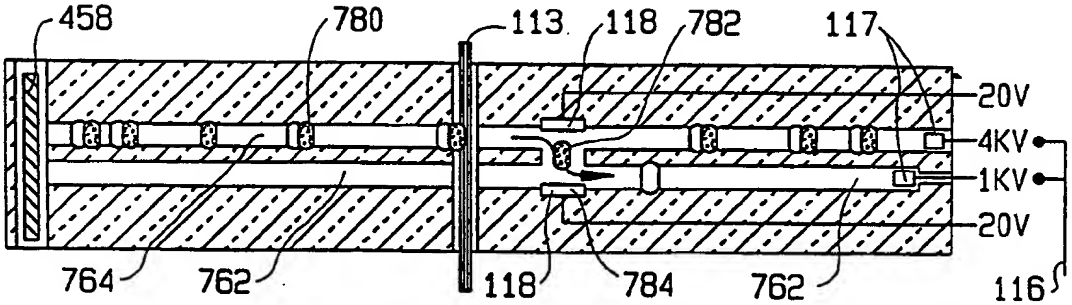


FIG. 7

O Normal ▨ Tumor

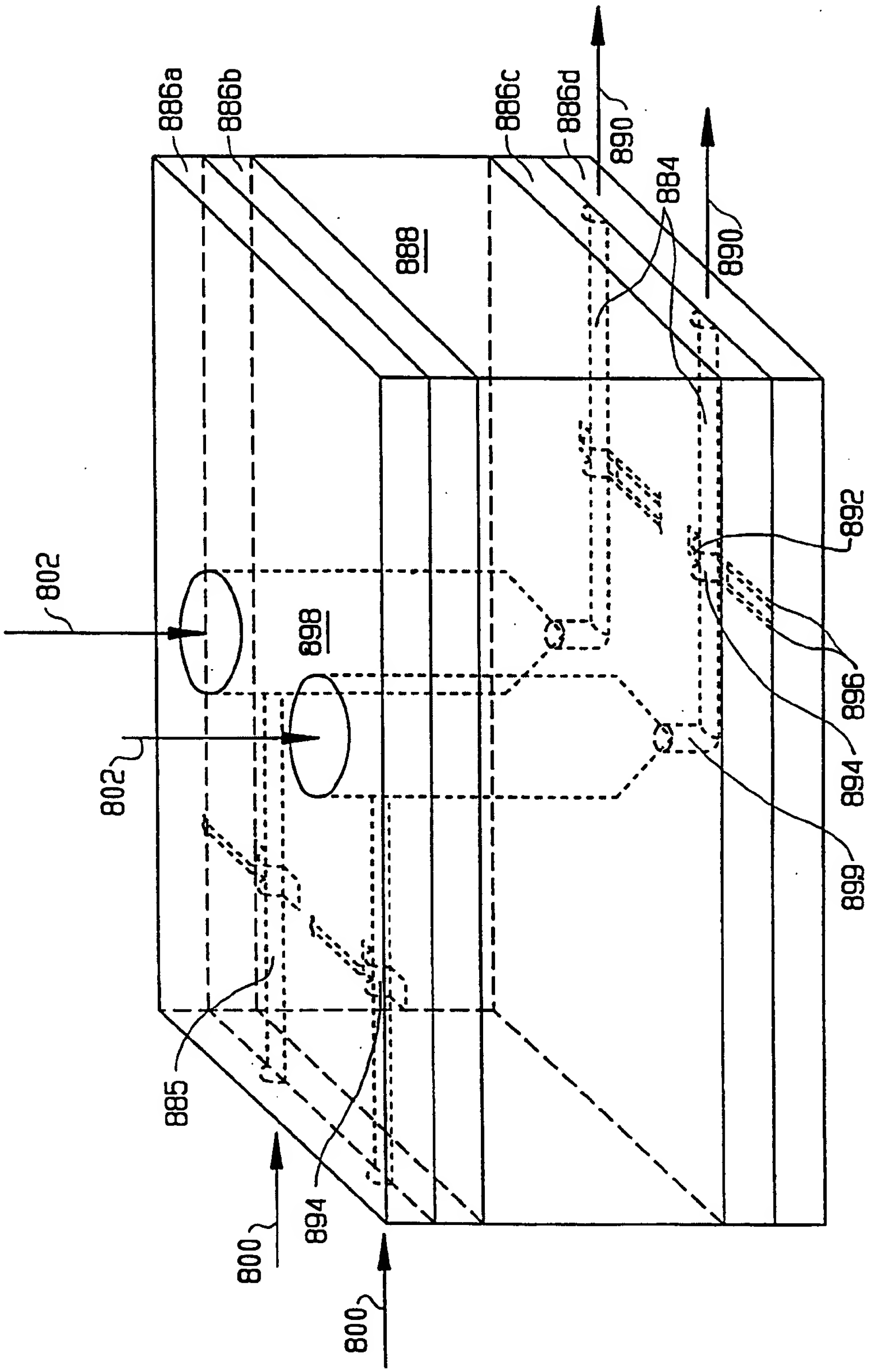

886a
886b
886c
886d
890
890
884
888
802
802
898
892
896
894
899
885
894
800
800

FIG. 8

FIG. 9A

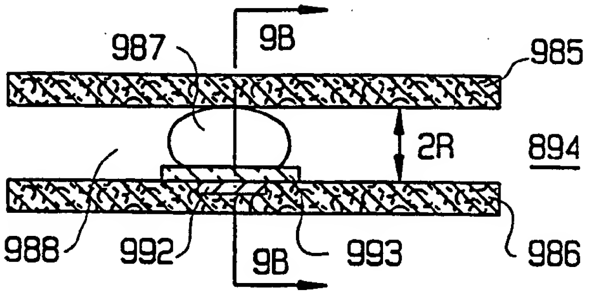


FIG. 9B

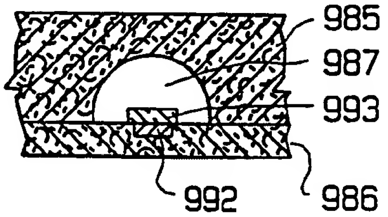


FIG. 10A

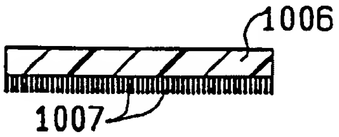


FIG. 10B

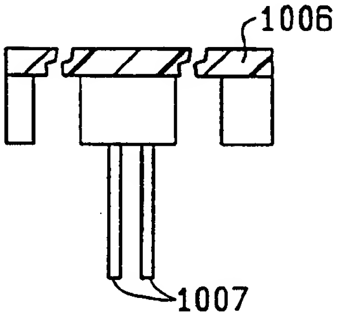


FIG. 10C

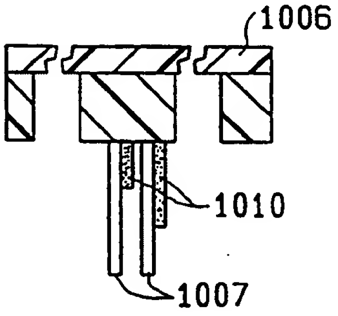
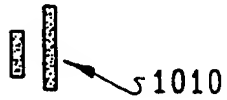


FIG. 10D



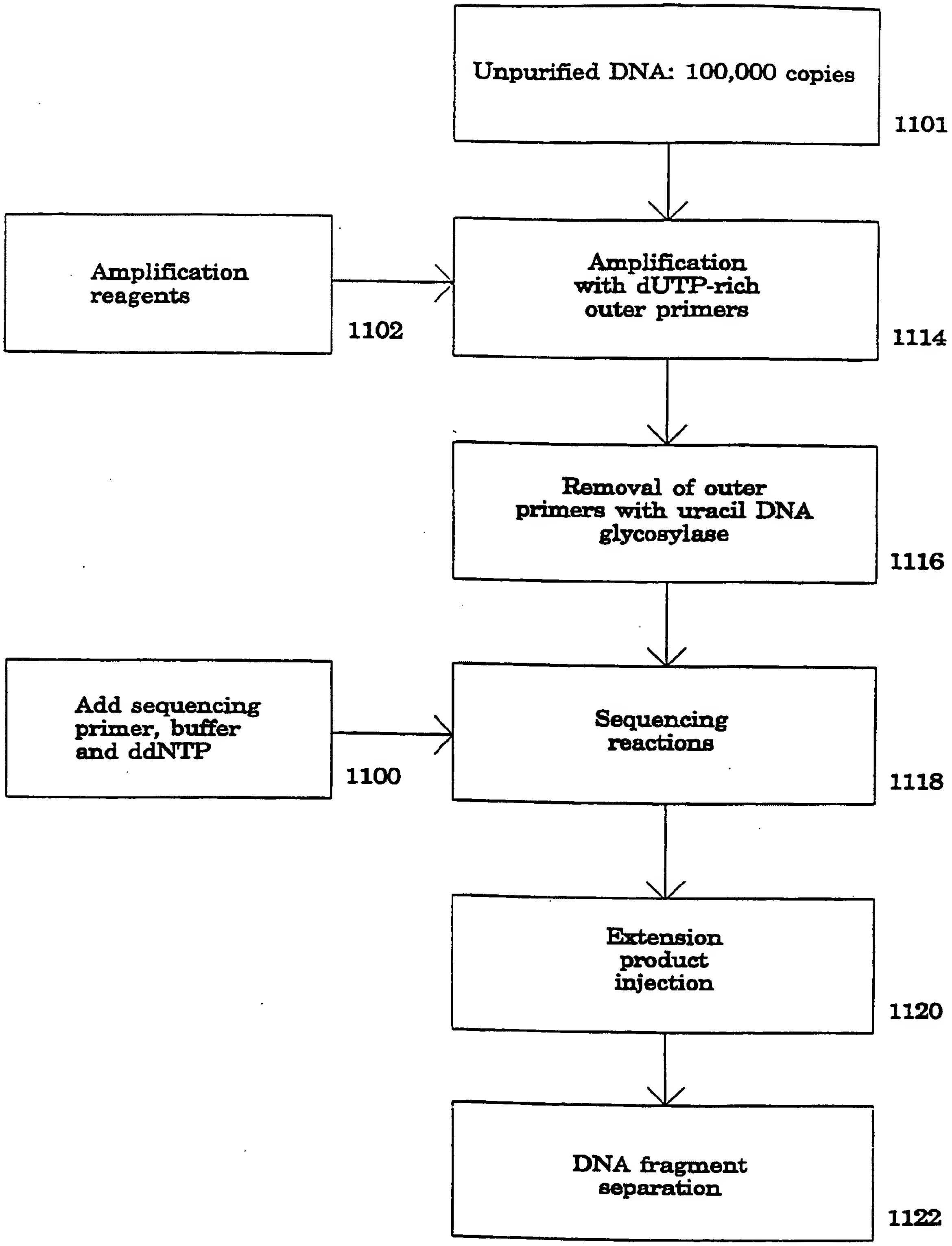

Unpurified DNA: 100,000 copies
1101
Amplification reagents
1102
Amplification with dUTP-rich outer primers
1114
Removal of outer primers with uracil DNA glycosylase
1116
Add sequencing primer, buffer and ddNTP
1100
Sequencing reactions
1118
Extension product injection
1120
DNA fragment separation
1122

Figure 11

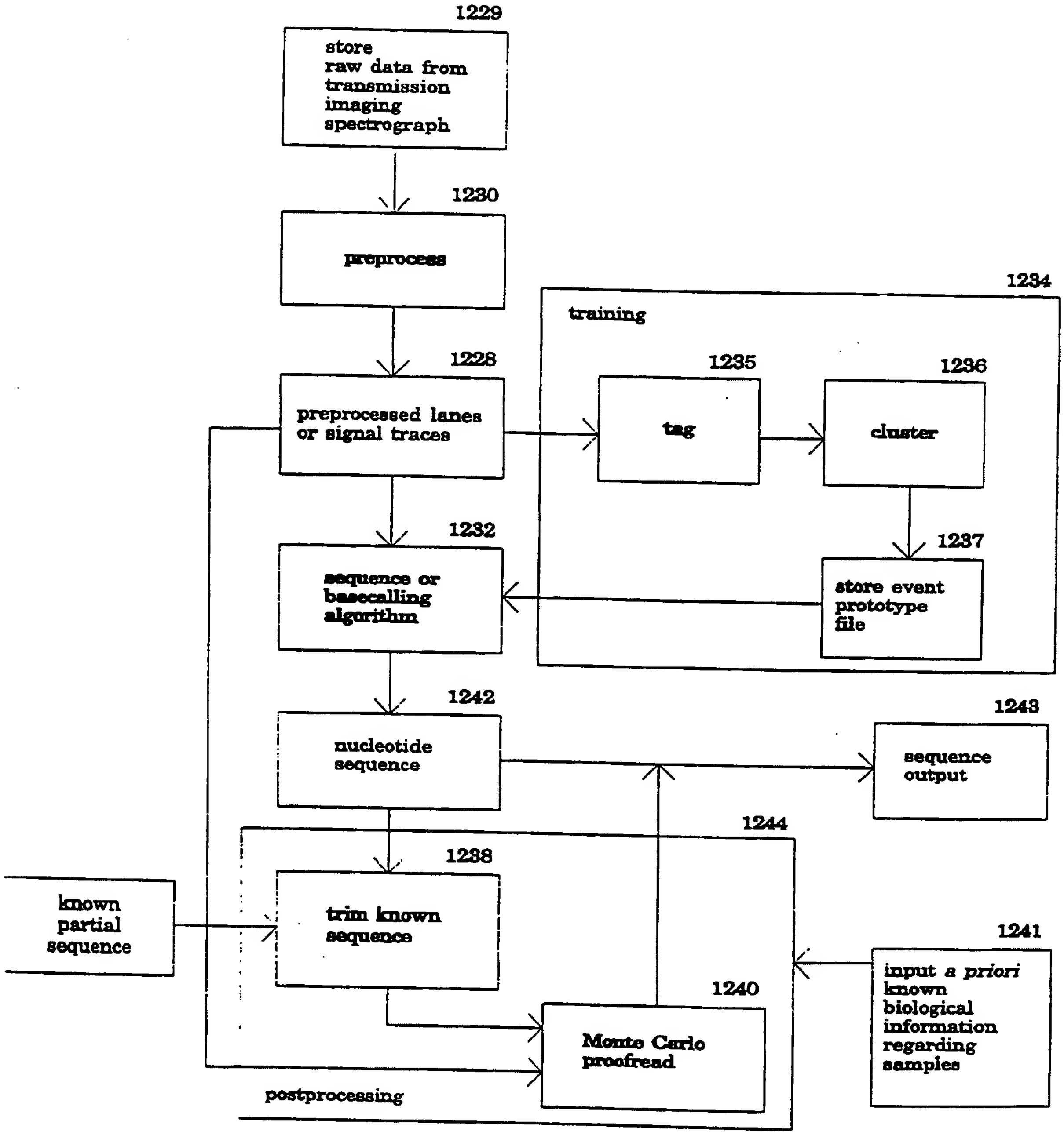

1229
store raw data from transmission imaging spectrograph
1230
preprocess
1234
training
1228
preprocessed lanes or signal traces
1235
tag
1236
cluster
1237
store event prototype file
1232
sequence or basecalling algorithm
1242
nucleotide sequence
1243
sequence output
1244
1238
known partial sequence
trim known sequence
1241
input a priori known biological information regarding samples
1240
Monte Carlo proofread
postprocessing

Figure 12

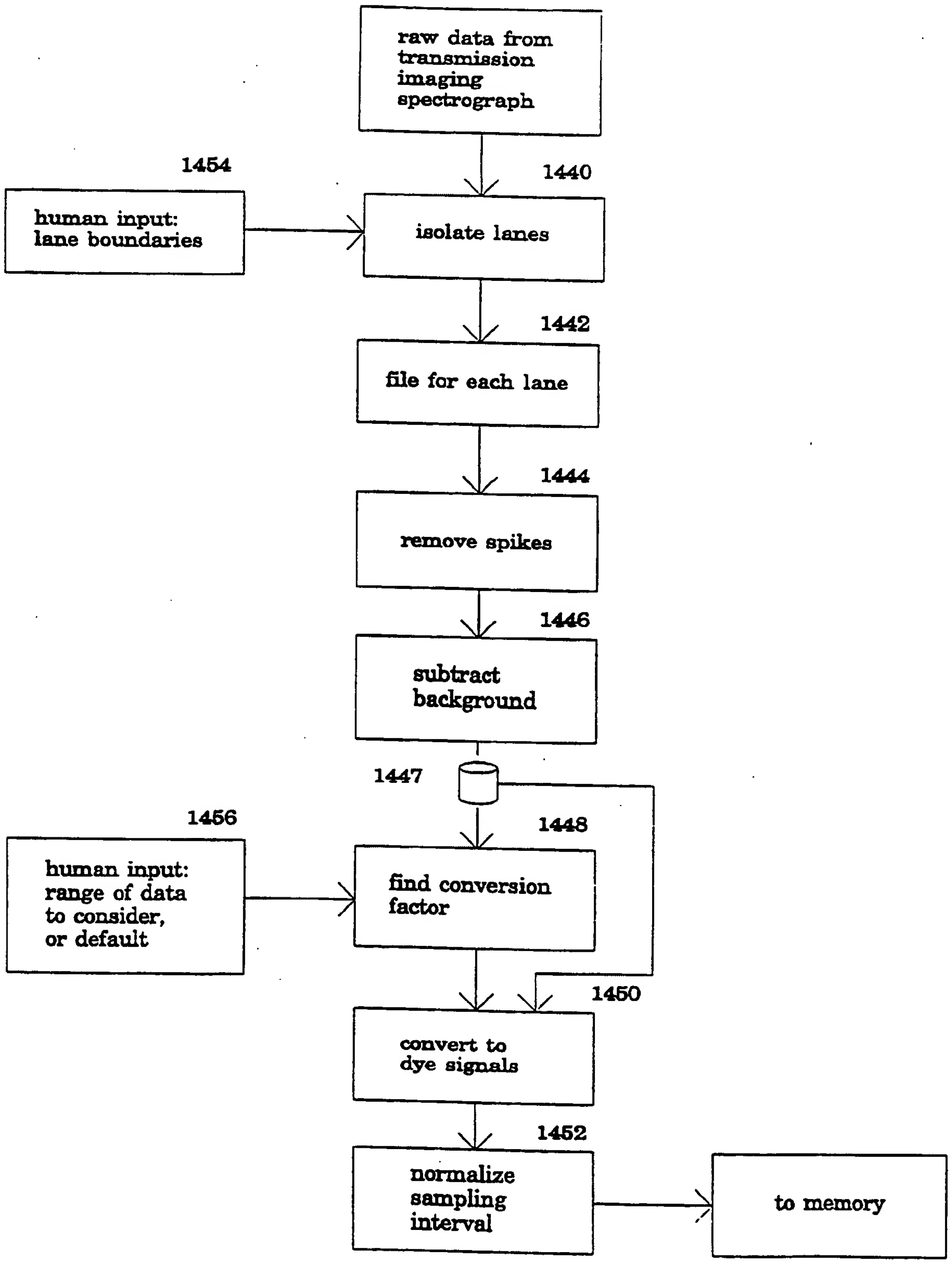

raw data from transmission imaging spectrograph
1454
1440
human input: lane boundaries
isolate lanes
1442
file for each lane
1444
remove spikes
1446
subtract background
1447
1448
1456
human input: range of data to consider, or default
find conversion factor
1450
convert to dye signals
1452
normalize sampling interval
to memory

Figure 13

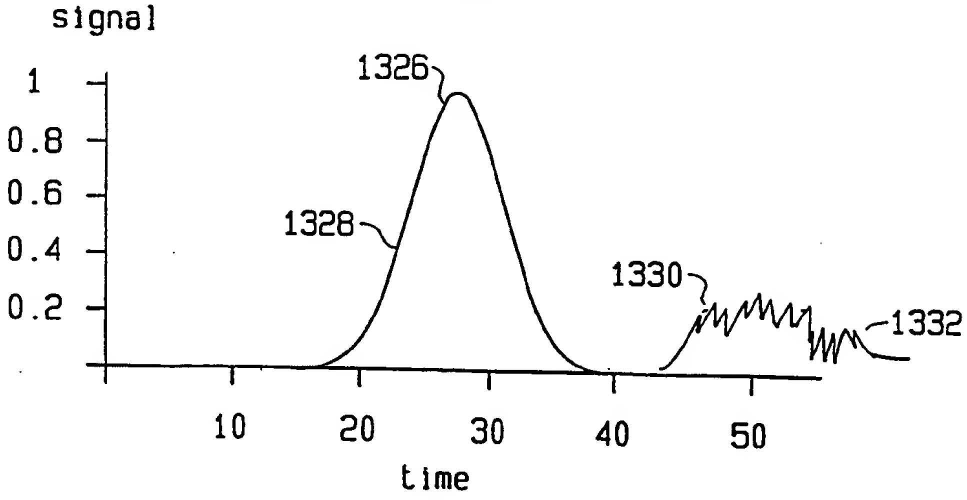


FIG. 14A

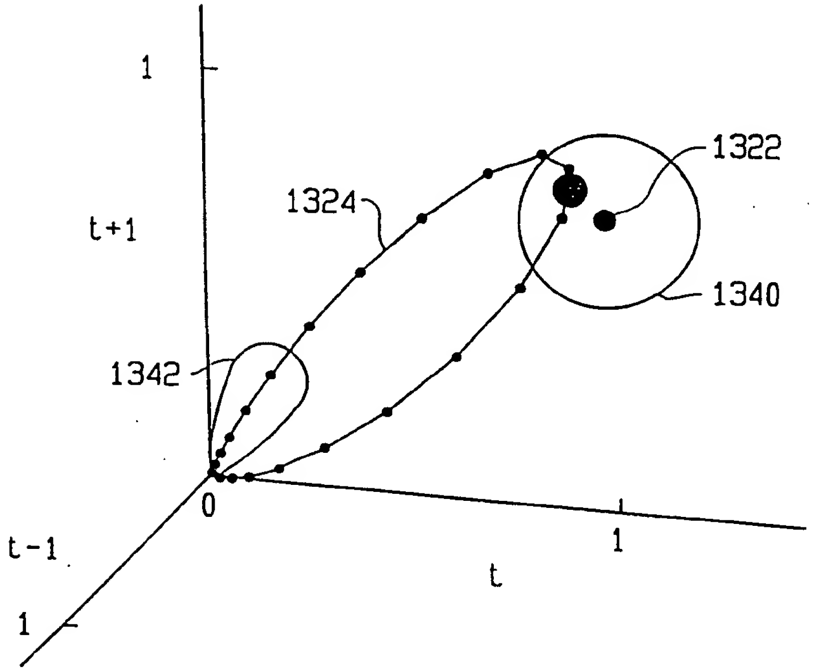


FIG. 14B

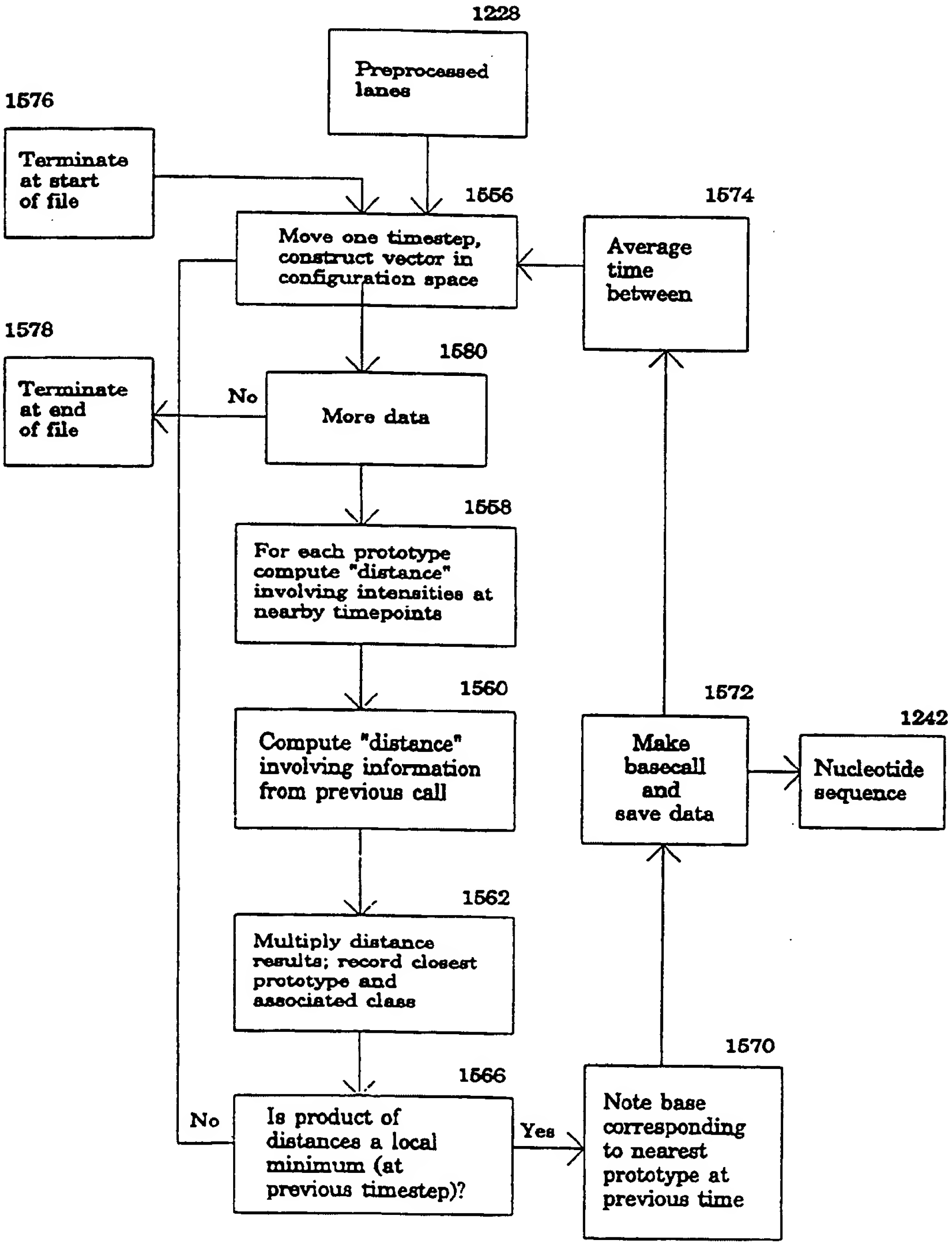

1228
Preprocessed lanes
1576
Terminate at start of file
1556
Move one timestep, construct vector in configuration space
1574
Average time between
1578
Terminate at end of file
1580
No
More data
1558
For each prototype compute "distance" involving intensities at nearby timepoints
1560
Compute "distance" involving information from previous call
1572
Make basecall and save data
1242
Nucleotide sequence
1562
Multiply distance results; record closest prototype and associated class
1570
1566
No
Is product of distances a local minimum (at previous timestep)?
Yes
Note base corresponding to nearest prototype at previous time

Figure 15

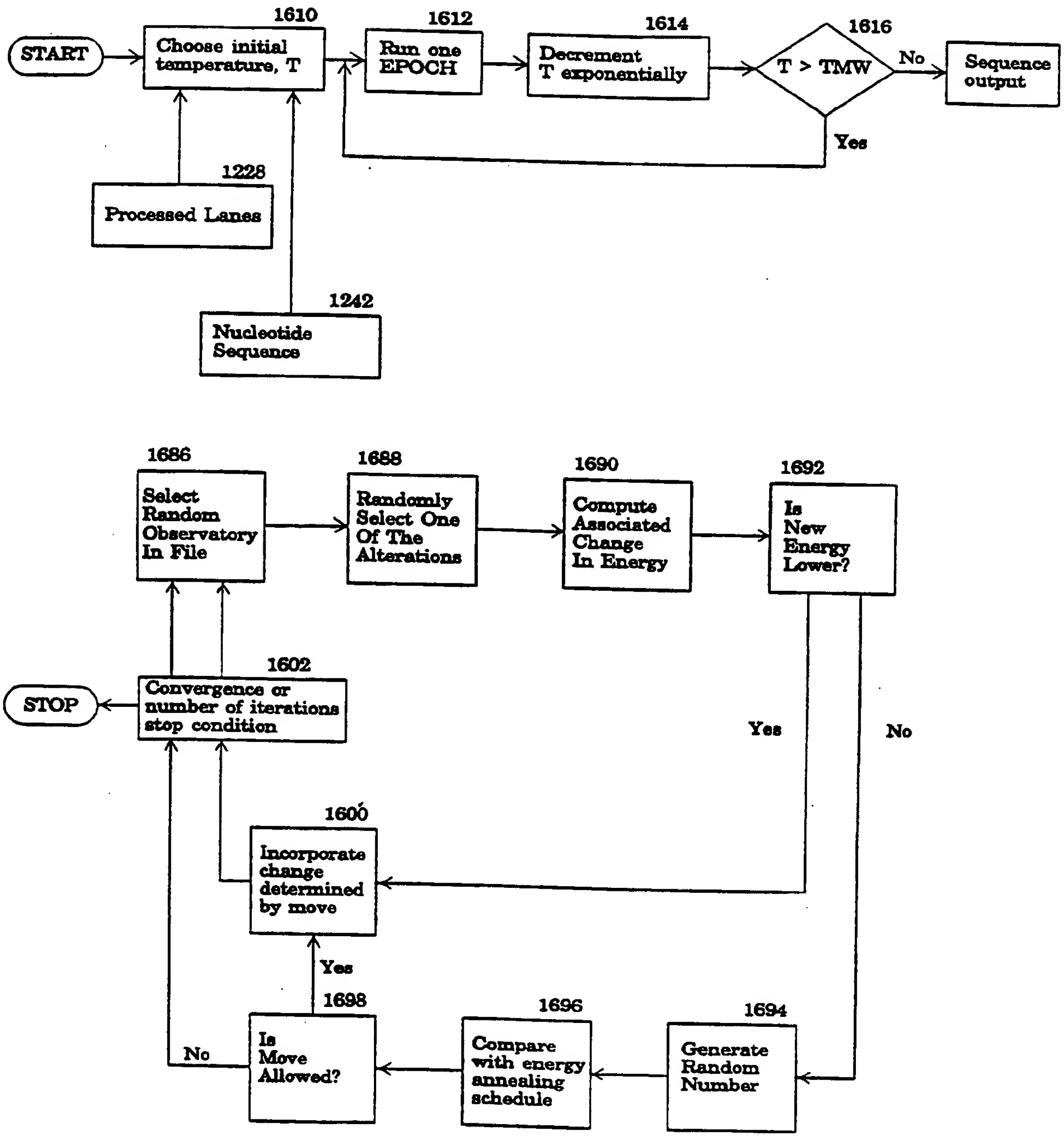

START
1610
Choose initial temperature, T
1612
Run one EPOCH
1614
Decrement T exponentially
1616
T > TMW
No
Sequence output
Yes
1228
Processed Lanes
1242
Nucleotide Sequence
1686
Select Random Observatory In File
1688
Randomly Select One Of The Alterations
1690
Compute Associated Change In Energy
1692
Is New Energy Lower?
Yes
No
1602
STOP
Convergence or number of iterations stop condition
1600
Incorporate change determined by move
Yes
1698
Is Move Allowed?
No
1696
Compare with energy annealing schedule
1694
Generate Random Number

Figure 16

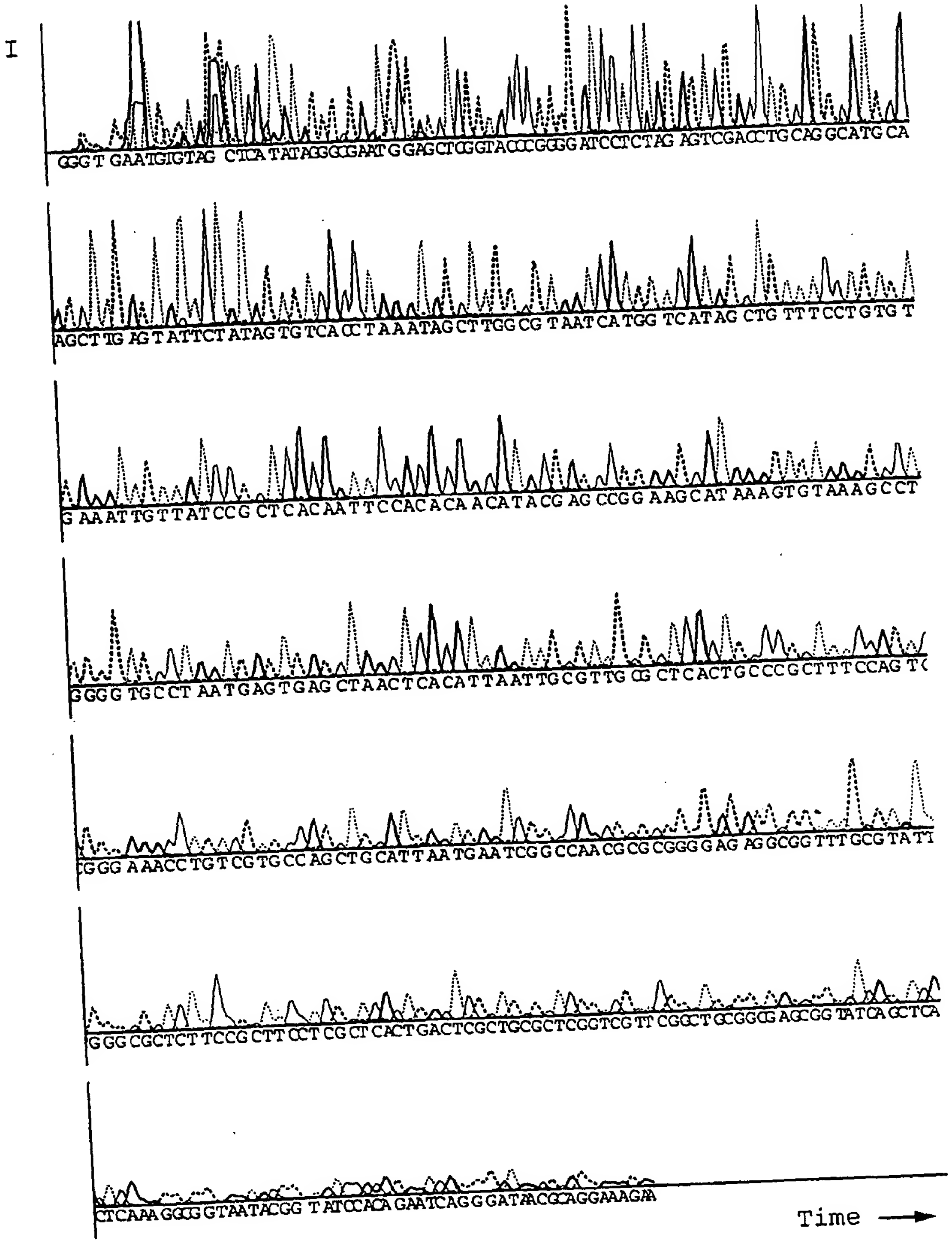

I
GGG T GAATGTGTAG CTCA TATAGGGCGAATG GAGCTCGGTACCCGGGG ATCCTCTAG AGTCGACCTG CAG GCATG CA
AGCT TG AG T ATTCT ATAG TG TCA CC T A A ATAG CT TGG CG T AAT CA TGG TC AT AG CTG TT T CCT G TG T
G AA AT TG T T AT CCG CT C AC AA T T CCAC A CA A CA T A CG AG CCGG A AG C AT AA A GT G T AA AG C CT
GGG G TG CCT AA T GAG T GAG CT AA CT CA CA TT AA TT GCG TTG CG CT CACT GCCCG CT TT CCAG T(
GGG A AACC TG T CG TG CC AG CT G CAT T AA TG AA TCG G CCAA CG CG CGGG G AG AG GCGG TTT GCG TA TT
G GG CGCTCT TCCG CTT CCT CG CT CACT GACTCGCTGCGCTCGGTCG TT CGGCT GCGGCG AGCGG TATCA GCTCA
CTCAAA GCGG GTAATACGG T ATCCACA GAATCAGG GATAACGCAGGAAAGAA
Time →

FIG. 17

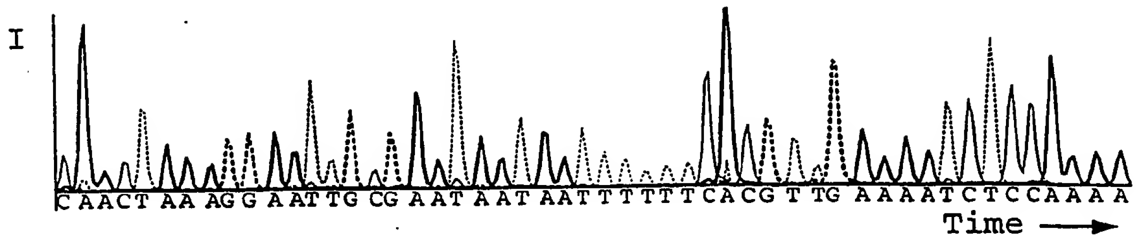


FIG. 18A

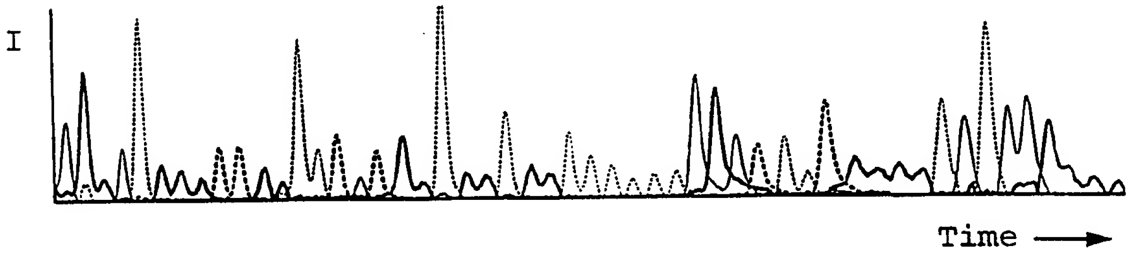


FIG. 18B

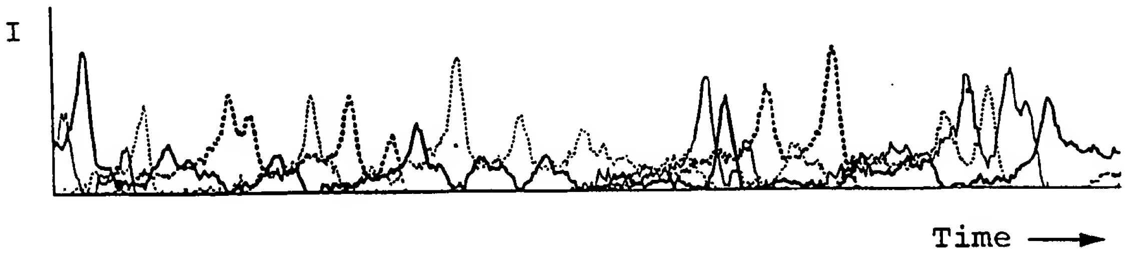


FIG. 18C

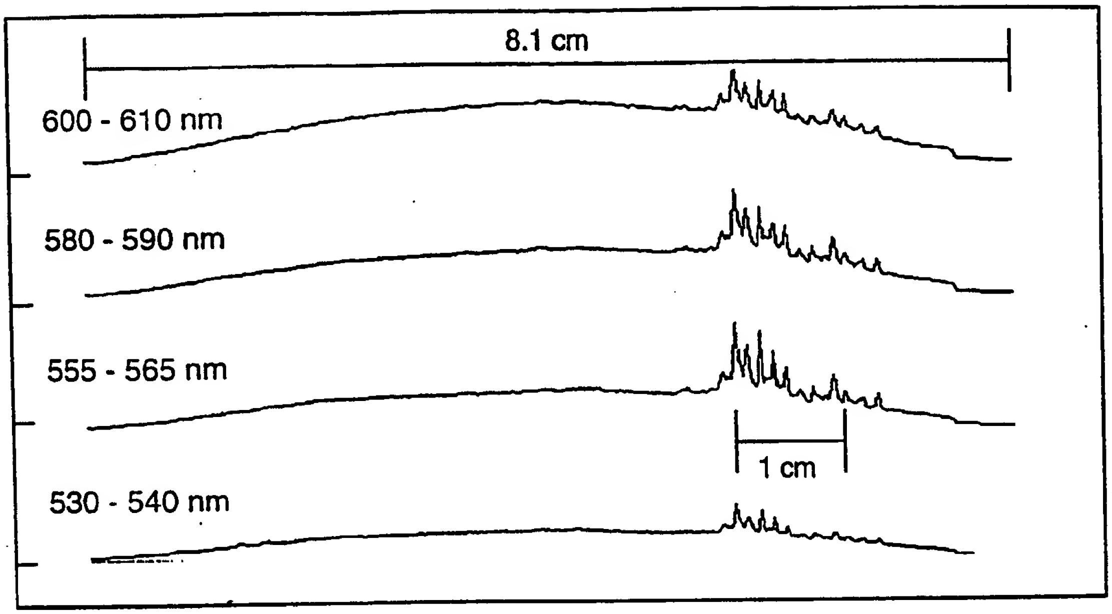


FIG. 19A

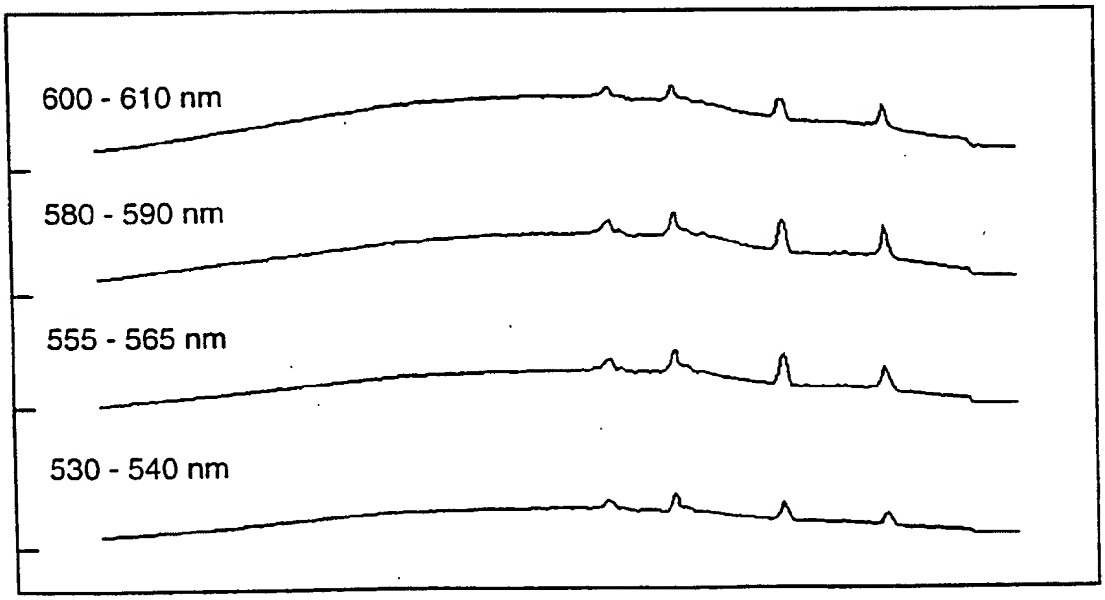


FIG. 19B

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US96/06579

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :Please See Extra Sheet.
US CL : 435/5, 6, 91.1, 91.2, 91.21; 204/182.8, 182.9, 186, 299R, 403
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 435/5, 6, 91.1, 91.2, 91.21; 204/182.8, 182.9, 186, 299R, 403

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

Please See Extra Sheet.

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X ---- Y	LAGERKVIST et al. Manifold sequencing: Efficient processing of large sets of sequencing reactions. Proc. Natl. Acad. Sci. March 1994, Vol. 91, pages 2245-2249.	1-6, 9-13, 15, 29-37, 41-50 ---------------- 7, 8, 14, 16-28, 38-40, 51-66
Y	HULTMAN et al. Direct solid phase sequencing of genomic and plasmid DNA using magnetic beads as solid support. Nucleic Acids Research. 1989, Vol. 17, No. 13, pages 4937-4946.	7, 65, 66
Y	US 5,149,416 A (OSTERHOUDT et al.) 22 September 1992, column 5, lines 24-26.	8, 65, 66

☒ Further documents are listed in the continuation of Box C. □ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be part of particular relevance		
"E"	earlier document published on or after the international filing date	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed	"&"	document member of the same patent family

Date of the actual completion of the international search

08 AUGUST 1996

Date of mailing of the international search report

05 SEP 1996

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231
Facsimile No. (703) 305-3230

Authorized officer
JEFFREY FREDMAN
Telephone No. (703) 308-0196

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US96/06579

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 5,066,382 A (WEINBERGER et al.) 19 November 1991, column 7, lines 33-48.	14, 65, 66
Y	HARRISON et al. Capillary electrophoresis and sample injection systems integrated on a planar glass chip. Anal. Chem. 1991, Vol. 64, pages 1926-1932, see entire document.	15-18, 20-24, 26, 38-40, 65, 66
Y	WOOLLEY et al. Ultra-high-speed DNA fragment separations using microfabricated capillary array electrophoresis chips. Proc. Natl. Acad. Sci. November 1994, Vol. 91, pages 11348-11252, see entire document.	15-18, 20-24, 26, 38-40, 51-66
Y	MATHIES et al. Capillary array electrophoresis: an approach to high-speed high-throughput DNA sequencing. Nature. Vol. 359, 10 September 1992, pages 167-169, see figure 1.	19, 65, 66
Y	HUBER et al, High-resolution liquid chromatography of DNA fragments on non-porous poly(styrene-divinylbenzene) particles. Nucleic Acids Research 1993, Vol. 21, No. 5, pages 1061-1066, see entire document.	28, 65, 66
X ---- Y	BALL et al. The use of uracil-N-glycosylase in the preparation of PCR products for direct sequencing. Nucleic Acids Research 1992, Vol. 20, No. 12, page 3255, see entire document.	51 --------- 52, 53
Y	KARGER et al. Multiwavelength fluorescence detection for DNA sequencing using capillary electrophoresis. Nucleic Acids Research, 1991, Vol. 19, No. 18, pages 4955-4962, see entire document.	54-64, 66
Y	SMITH et al. Quantitative analysis of one-dimensional gel electrophoresis profiles. CABIOS. 1990, Vol. 6, No. 2, pages 93-99, see entire document.	54-64, 66
Y	AHN et al, A fully integrated micromachined magneitc particle manipulator and separator. J. Micromechanical Syst. 1993, Vol. 2, No. 1, pages 15-22, see entire document.	7, 25
Y	JERMAN. 'Electrically activated, normally closed diaphragm valves'. In: International Conference on Solid State Sensors, 1991, see entire document. I	39
Y	LIN et al. 'Microbubble powered actuator'. In: International Conference on Solid State Sensors, and Actuators, Transducers, June 1991, pages 1041-1044.	39

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	TRAUB. Constant-dispersion gisrm spectrometer for channeled spectra. J. Opt. Soc. Am. A. September 1990, Vol. 7, No. 9, pages 1779-1791, see entire document.	29-34
Y	LOEWEN et al. Grating efficiency theory as it applies to blazed and holographic gratings. Applied Optics. October 1977, Vol. 16, No. 10, pages 2711-2721, see entire document.	29-34
Y	EP 0 376 611 A2 (THE BOARD OF TRUSTEES OF THE LELAND STANFORD JUNIOR UNIVERSITY) 04 July 1990, see entire document.	1-66
A	LIANG et al. Distribution and cloning of eukaryotic mRNAs by means of differential display: refinements and optimization. Nucleic Acids Research, 1993, Vol. 21, No. 14, pages 3269-3275, see entire document.	51-64
A	SWERDLOW et al. Capillary gel electrophoresis for rapid, high resolution DNA sequencing. Nucleic Acids Research, 1990, Vol. 18, No. 6, pages 1415-1419, see entire document.	1-66
A	COMPTON et al. Capillary Electrophoresis. Biotechniques. 1988, Vol. 6, No. 5, pages 432-439, see entire document.	1-66
A	KUHR et al. Capillary Electrophoresis. Anal. Chem. 1992, Vol. 64, pages 389R-407R, see entire document.	1-66

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US96/06579

Box I Observations where certain claims were found unsearchable (Continuation of item 1 of first sheet)

This international report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. □ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. □ Claims Nos.:
because they relate to parts of the international application that do not comply with the prescribed requirements to such an extent that no meaningful international search can be carried out, specifically:

3. □ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box II Observations where unity of invention is lacking (Continuation of item 2 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

Please See Extra Sheet.

1. □ As all required additional search fees were timely paid by the applicant, this international search report covers all searchable claims.

2. ☒ As all searchable claims could be searched without effort justifying an additional fee, this Authority did not invite payment of any additional fee.

3. □ As only some of the required additional search fees were timely paid by the applicant, this international search report covers only those claims for which fees were paid, specifically claims Nos.:

4. □ No required additional search fees were timely paid by the applicant. Consequently, this international search report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

Remark on Protest □ The additional search fees were accompanied by the applicant's protest.
□ No protest accompanied the payment of additional search fees.

A. CLASSIFICATION OF SUBJECT MATTER:
IPC (6):

C12Q 1/68, 1/70; C12P 19/34; C07H 21/02, 21/04; C25B 1/00, 7/00; B01D 61/42, 61/44; C25D 13/00; G01N 27/26

B. FIELDS SEARCHED
Electronic data bases consulted (Name of data base and where practicable terms used):

APS, MEDLINE, BIOSIS, CAPLUS, WPIDS
search terms: sequencer, sequencing, automat?, machin?, electrophor?, nucleic, DNA, RNA, protein, solid, phase, loading, comb, separat?, denatur?, adhesion, plate, glass, Peltier, grooves, etching, capillary, controller, dUTP, degrade, PCR, polymerase, chain

BOX II. OBSERVATIONS WHERE UNITY OF INVENTION WAS LACKING
This ISA found multiple inventions as follows:

This application contains the following inventions or groups of inventions which are not so linked as to form a single inventive concept under PCT Rule 13.1. In order for all inventions to be examined, the appropriate additional examination fees must be paid.

Group I, claim(s) 1-50, 65 and 66, drawn to an apparatus for preparation of biopolymer fragment samples.
Group II, claim(s) 51-53, drawn to a method for generating reaction fragments.
Group III, claim(s) 54-64, drawn to a method for sequencing DNA.

The inventions listed as Groups I-III do not relate to a single inventive concept under PCT Rule 13.1 because, under PCT Rule 13.2, they lack the same or corresponding special technical features for the following reasons: The apparatus of Group I, which is designed to separate biofragments is not specially adapted in any way to perform the methods of either Group II or Group III. The apparatus can be used to separate the nucleic acids of Groups II or III but it could also be used in a variety of other nucleic acid methods as well as methods of separation of polypeptides, lipids or carbohydrates. The methods of Groups II or III are not specially adapted to be performed together or with the apparatus of Group I and could be performed "by hand" as well as in a variety of other apparatus..